Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapter 1:

Software and Software Engineering

1.1 The Nature of Software...

- Software is intangible
  - Hard to understand development effort
- Software is easy to reproduce
  - Cost is in its *development*
    - —in other engineering products, manufacturing is the costly stage
- The industry is labor-intensive
  - Hard to automate

The Nature of Software ...

- Untrained people can hack something together
  - Quality problems are hard to notice
- Software is easy to modify
  - People make changes without fully understanding it
- Software does not 'wear out'
  - It deteriorates by having its design changed:
    - —erroneously, or
    - —in ways that were not anticipated, thus making it complex

The Nature of Software

- Conclusions
  - Much software has poor design and is getting worse
  - Demand for software is high and rising
  - We are in a perpetual 'software crisis'
  - We have to learn to 'engineer' software

Types of Software...

- Custom
  - For a specific customer
- Generic
  - Sold on open market
  - Often called
    - —COTS (Commercial Off The Shelf)
    - —Shrink-wrapped
- Embedded
  - Built into hardware
  - Hard to change

Types of Software

Differences among custom, generic and embedded software

	Custom	Generic	Embedded
Number of *copies* in use	low	medium	high
Total *processing power* devoted to running this type of software	low	high	medium
Worldwide annual *development effort*	high	medium	low

Types of Software

- Real time software
  - E.g. control and monitoring systems
  - Must react immediately
  - Safety often a concern
- Data processing software
  - Used to run businesses
  - Accuracy and security of data are key

Some software has both aspects

1.2 What is Software Engineering?...

The process of solving customers' problems by the systematic development and evolution of large, high-quality software systems within cost, time and other constraints

Solving customers' problems
- This is the goal of software engineering
- Sometimes the solution is to buy, not build
- Adding unnecessary features does not help solve the problem
- Software engineers must communicate effectively to identify and understand the problem

What is Software Engineering?…

Systematic development and evolution

- An engineering process involves applying well understood techniques in a organized and disciplined way
- Many well-accepted practices have been formally standardized
  —e.g. by the IEEE or ISO
- Most development work is *evolution*

Large, high quality software systems

- Software engineering techniques are needed because large systems cannot be completely understood by one person
- Teamwork and co-ordination are required
- Key challenge: Dividing up the work and ensuring that the parts of the system work properly together
- The end-product that is produced must be of sufficient quality

What is Software Engineering?

Cost, time and other constraints

- Finite resources
- The benefit must outweigh the cost
- Others are competing to do the job cheaper and faster
- Inaccurate estimates of cost and time have caused many project failures

1.3 Software Engineering and the Engineering Profession

The term Software Engineering was coined in 1968

- People began to realize that the principles of engineering should be applied to software development

Engineering is a licensed profession

- In order to protect the public
- Engineers design artifacts following well accepted practices which involve the application of science, mathematics and economics
- Ethical practice is also a key tenet of the profession

1.4 Stakeholders in Software Engineering

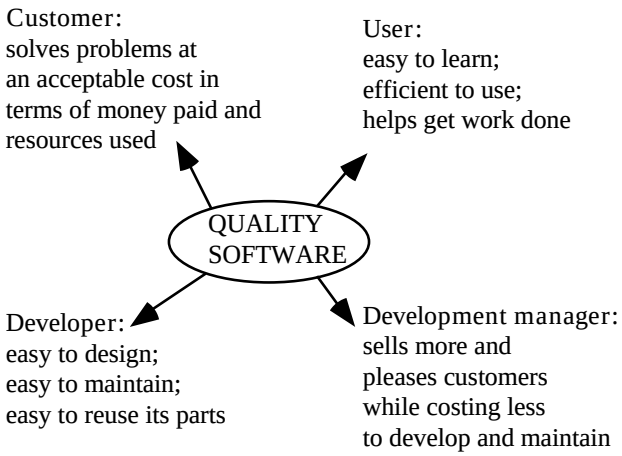
1. Users
   - Those who use the software
2. Customers
   - Those who pay for the software
3. Software developers
4. Development Managers

All four roles can be fulfilled by the same person

1.5 Software Quality...

- Usability
  - Users can learn it and fast and get their job done easily
- Efficiency
  - It doesn't waste resources such as CPU time and memory
- Reliability
  - It does what it is required to do without failing
- Maintainability
  - It can be easily changed
- Reusability
  - Its parts can be used in other projects, so reprogramming is not needed

Software Quality...



Software Quality

- The different qualities can conflict
  - Increasing efficiency can reduce maintainability or reusability
  - Increasing usability can reduce efficiency
- Setting objectives for quality is a key engineering activity
  - You then design to meet the objectives
  - Avoids 'over-engineering' which wastes money
- Optimizing is also sometimes necessary
  - E.g. obtain the highest possible reliability using a fixed budget

Internal Quality Criteria

These:

- Characterize aspects of the design of the software
- Have an effect on the external quality attributes
- E.g.
  - —The amount of commenting of the code
  - —The complexity of the code

Short Term Vs. Long Term Quality

Short term:

- Does the software meet the customer's immediate needs?
- Is it sufficiently efficient for the volume of data we have today?

Long term:

- Maintainability
- Customer's future needs

1.6 Software Engineering Projects

Most projects are evolutionary or maintenance projects, involving work on *legacy* systems

- Corrective projects: fixing defects
- Adaptive projects: changing the system in response to changes in
  - —Operating system
  - —Database
  - —Rules and regulations
- Enhancement projects: adding new features for users
- Reengineering or perfective projects: changing the system internally so it is more maintainable

Software Engineering Projects

'Green field' projects

- New development
- The minority of projects

Software Engineering Projects

Projects that involve building on a *framework* or a set of existing components.

- The framework is an application that is missing some important details.
  - —E.g. Specific rules of this organization.
- Such projects:
  - —Involve plugging together *components* that are:
    - Already developed.
    - Provide significant functionality.
  - —Benefit from reusing reliable software.
  - —Provide much of the same freedom to innovate found in green field development.

1.7 Activities Common to Software Projects...

Requirements and specification

Includes

—Domain analysis

—Defining the problem

—Requirements gathering

- Obtaining input from as many sources as possible

—Requirements analysis

- Organizing the information

—Requirements specification

- Writing detailed instructions about how the software should behave

Activities Common to Software Projects...

Design

Deciding how the requirements should be implemented, using the available technology

Includes:

—Systems engineering: Deciding what should be in hardware and what in software

—Software architecture: Dividing the system into subsystems and deciding how the subsystems will interact

—Detailed design of the internals of a subsystem

—User interface design

—Design of databases

Activities Common to Software Projects

Modeling

Creating representations of the domain or the software

—Use case modeling

—Structural modeling

—Dynamic and behavioural modeling

Programming

Quality assurance

Reviews and inspections

Testing

Deployment

Managing the process

1.8 The Eight Themes of the Book

1. Understanding the customer and the user
2. Basing development on solid principles and reusable technology
3. Object orientation
4. Visual modeling using UML
5. Evaluation of alternatives
6. Iterative development
7. Communicating effectively using documentation
8. Risk management in all SE activities

1.9 Difficulties and Risks in Software Engineering

- Complexity and large numbers of details
- Uncertainty about technology
- Uncertainty about requirements
- Uncertainty about software engineering skills
- Constant change
- Deterioration of software design
- Political risks

Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapter 2:

Review of Object Orientation

2.1 What is Object Orientation?

Procedural paradigm:

Software is organized around the notion of *procedures*

Procedural abstraction

—Works as long as the data is simple

Adding data abstractions

—Groups together the pieces of data that describe some entity

—Helps reduce the system's complexity.

- Such as *Records* and *structures*

Object oriented paradigm:

Organizing procedural abstractions in the context of data abstractions

Object Oriented paradigm

An approach to the solution of problems in which all computations are performed in the context of objects.

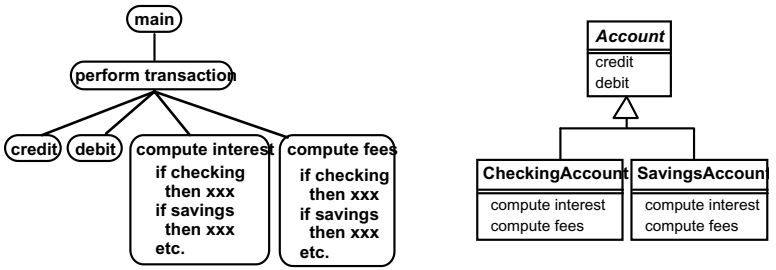
The objects are instances of classes, which:

—are data abstractions

—contain procedural abstractions that operation on the objects

A running program can be seen as a collection of objects collaborating to perform a given task

A View of the Two paradigms



2.2 Classes and Objects

Object

A chunk of structured data in a running software system

Has *properties*

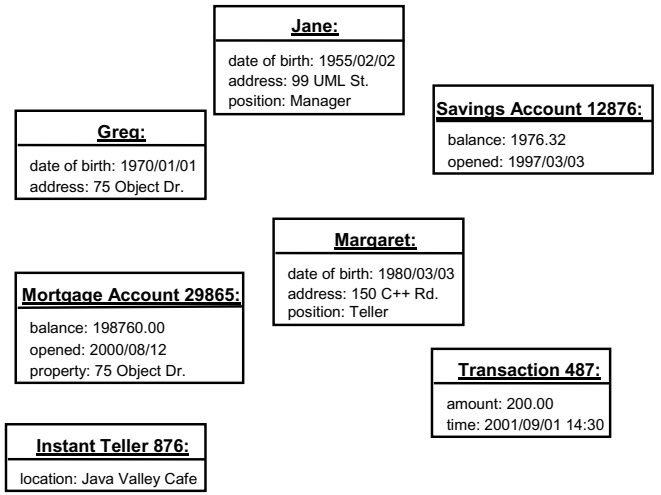
—Represent its state

Has *behaviour*

—How it acts and reacts

—May simulate the behaviour of an object in the real world

Objects



Classes

A class:

Is a unit of abstraction in an object oriented (OO) program

Represents similar objects

—Its *instances*

Is a kind of software module

—Describes its instances' structure (properties)

—Contains *methods* to implement their behaviour

Is Something a Class or an Instance?

Something should be a *class* if it could have instances

Something should be an *instance* if it is clearly a *single* member of the set defined by a class

Film

Class; instances are individual films.

Reel of Film:

Class; instances are physical reels

Film reel with serial number SW19876

Instance of **`ReelOfFilm`**

Science Fiction

Instance of the class **`Genre`**.

Science Fiction Film

Class; instances include 'Star Wars'

Showing of 'Star Wars' in the Phoenix Cinema at 7 p.m.:

Instance of **`ShowingOfFilm`**

Naming classes

Use *capital* letters
—E.g. `BankAccount` not `bankAccount`

Use *singular* nouns

Use the right level of generality
—E.g. `Municipality`, not `City`

Make sure the name has only *one* meaning
—E.g. 'bus' has several meanings

2.3 Instance Variables

Variables defined inside a class corresponding to data present in each instance

Attributes
—Simple data
—E.g. `name`, `dateOfBirth`

Associations
—Relationships to other important classes
—E.g. `supervisor`, `coursesTaken`
—More on these in Chapter 5

Variables vs. Objects

A variable
Refers to an object
May refer to different objects at different points in time

An object can be referred to by several different variables at the same time

Type of a variable
Determines what classes of objects it may contain

Class variables

A *class variable's* value is *shared* by all instances of a class.
Also called a *static* variable

If one instance sets the value of a class variable, then all the other instances see the same changed value.

Class variables are useful for:
—Default or 'constant' values (e.g. PI)
—Lookup tables and similar structures

Caution: *do not over-use class variables*

2.4 Methods, Operations and Polymorphism

Operation

- A higher-level procedural abstraction that specifies a type of behaviour
- Independent of any code which implements that behaviour
  - —E.g., calculating area (in general)

Methods, Operations and Polymorphism

Method

- A procedural abstraction used to implement the behaviour of a class.
- Several different classes can have methods with the same name
  - —They implement the same abstract operation in ways suitable to each class
  - —E.g, calculating area in a rectangle is done differently from in a circle

Polymorphism

A property of object oriented software by which an abstract operation may be performed in different ways in different classes.

- Requires that there be multiple methods of the same name
- The choice of which one to execute depends on the object that is in a variable
- Reduces the need for programmers to code many `if-else` or `switch` statements

2.5 Organizing Classes into Inheritance Hierarchies

Superclasses

- Contain features common to a set of subclasses

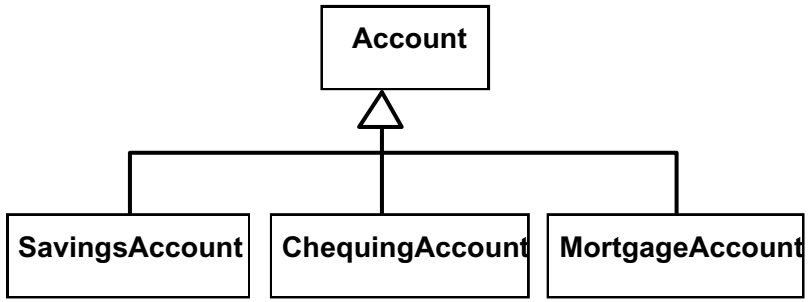
Inheritance hierarchies

- Show the relationships among superclasses and subclasses
- A triangle shows a *generalization*

Inheritance

- The *implicit* possession by all subclasses of features defined in its superclasses

An Example Inheritance Hierarchy



Inheritance

The *implicit* possession by all subclasses of features defined in its superclasses

The Isa Rule

Always check generalizations to ensure they obey the isa rule

"A checking account *is an* account"

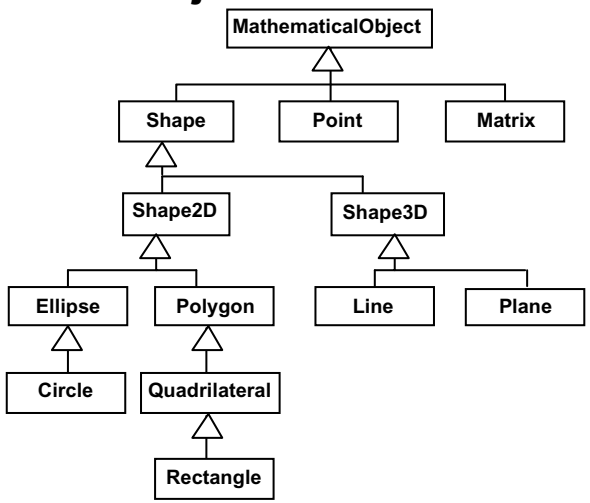
"A village *is a* municipality"

Should 'Province' be a subclass of 'Country'?

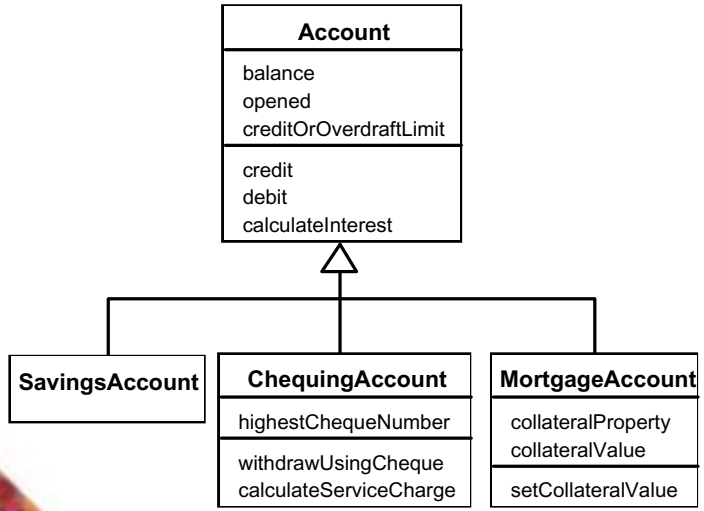
No, it violates the isa rule

—"A province *is a* country" is invalid!

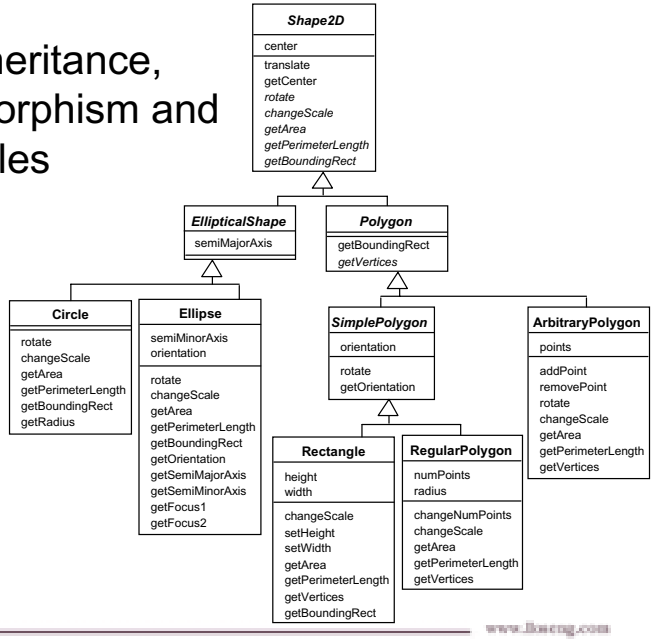
A possible inheritance hierarchy of mathematical objects



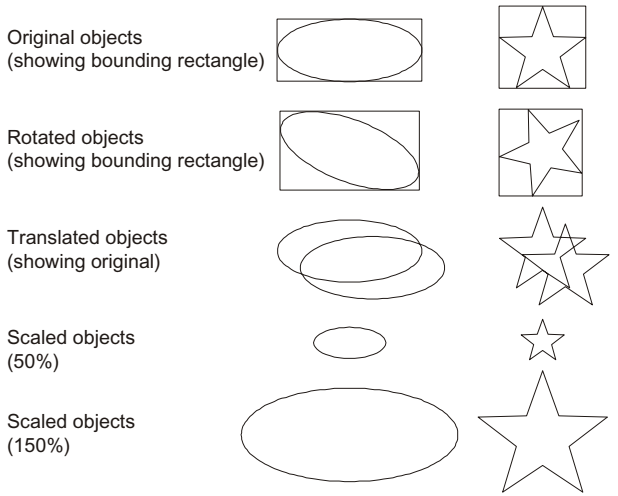
Make Sure all Inherited Features Make Sense in Subclasses



2.6 Inheritance, Polymorphism and Variables



Some Operations in the Shape Example



Abstract Classes and Methods

An operation should be declared to exist at the highest class in the hierarchy where it makes sense

- The operation may be *abstract* (lacking implementation) at that level
- If so, the class also must be *abstract*
  - —No instances can be created
  - —The opposite of an abstract class is a *concrete* class
- If a superclass has an abstract operation then its subclasses at some level must have a concrete method for the operation
  - —Leaf classes must have or inherit concrete methods for all operations
  - —Leaf classes must be concrete

Overriding

A method would be inherited, but a subclass contains a new version instead

- For restriction
  - —E.g. `scale(x,y)` would not work in `Circle`
- For extension
  - —E.g. `SavingsAccount` might charge an extra fee following every debit
- For optimization
  - —E.g. The `getPerimeterLength` method in `Circle` is much simpler than the one in `Ellipse`

Immutable objects

Instance variables may only be set when an object is first created.

None of the operations allow any changes to the instance variables

—E.g. a `scale` method could only create a new object, not modify an existing one

How a decision is made about which method to run

1. If there is a concrete method for the operation in the current class, run that method.
2. Otherwise, check in the immediate superclass to see if there is a method there; if so, run it.
3. Repeat step 2, looking in successively higher superclasses until a concrete method is found and run.
4. If no method is found, then there is an error

   In Java and C++ the program would not have compiled

Dynamic binding

Occurs when decision about which method to run can only be made at *run time*

Needed when:

—A variable is declared to have a superclass as its type, and

—There is more than one possible polymorphic method that could be run among the type of the variable and its subclasses

2.7 Concepts that Define Object Orientation

Necessary for a system or language to be OO

Identity

—Each object is distinct from each other object, and can be referred to

—Two objects are distinct even if they have the same data

Classes

—The code is organized using classes, each of which describes a set of objects

Inheritance

—The mechanism where features in a hierarchy inherit from superclasses to subclasses

Polymorphism

—The mechanism by which several methods can have the same name and implement the same abstract operation.

Other Key Concepts

- Abstraction
  - Object -> something in the world
  - Class -> objects
  - Superclass -> subclasses
  - Operation -> methods
  - Attributes and associations -> instance variables
- Modularity
  - Code can be constructed entirely of classes
- Encapsulation
  - Details can be hidden in classes
  - This gives rise to *information hiding*:
    - —Programmers do not need to know all the details of a class

The Basics of Java

- History
  - The first object oriented programming language was Simula-67
    - —designed to allow programmers to write simulation programs
  - In the early 1980's, Smalltalk was developed at Xerox PARC
    - —New syntax, large open-source library of reusable code, bytecode, platform independence, garbage collection.
  - late 1980's, C++ was developed by B. Stroustrup,
    - —Recognized the advantages of OO but also recognized that there were tremendous numbers of C programmers
  - In 1991, engineers at Sun Microsystems started a project to design a language that could be used in consumer 'smart devices': Oak
    - —When the Internet gained popularity, Sun saw an opportunity to exploit the technology.
    - —The new language, renamed Java, was formally presented in 1995 at the SunWorld '95 conference.

Java documentation

- Looking up classes and methods is an essential skill
  - Looking up unknown classes and methods will get you a long way towards understanding code
- Java documentation can be automatically generated by a program called Javadoc
  - Documentation is generated from the code and its comments
  - You should format your comments as shown in some of the book's examples
    - —These may include embeded html

Overview of Java

- The next few slides will remind you of several key Java features
  - Not in the book
  - See the book's web site for
    - —A more detailed overview of Java
    - —Pointers to tutorials, books etc.

Characters and Strings

`Character` is a class representing Unicode characters

- More than a byte each
- Represent any world language

`char` is a primitive data type containing a Unicode character

`String` is a class containing collections of characters

- + is the operator used to concatenate strings

Arrays and Collections

Arrays are of fixed size and lack methods to manipulate them

`Vector` is the most widely used class to hold a collection of other objects

- More powerful than arrays, but less efficient

`Iterator`s are used to access members of **`Vector`**s

- `Enumeration`s were formally used, but were more complex

```
v = new Vector();

Iterator i = v.iterator();
while(i.hasNext())
{
  aMethod(v.next());
}
```


Casting

Java is very strict about types

- If a variable is declared to have the type X, you can only invoke operations on it that are defined in class X or its superclasses
  - —Even though an instance of a *subclass* of X may be actually stored in the variable
- If you *know* an instance of a subclass is stored, then you can *cast* the variable to the subclass
  - —E.g. if I know a `Vector` contains instances of `String`, I can get the next element of its `Iterator` using:

```
(String)iterator.next();
```


Exceptions

Anything that can go wrong should result in the raising of an Exception

- `Exception` is a class with many subclasses for specific things that can go wrong

Use a try - catch block to trap an exception

```
try
{
  // some code
}
catch (ArithmeticException e)
{
  // code to handle division by zero
}
```

Interfaces

- Like abstract classes, but cannot have executable statements
  - Define a set of operations that make sense in several classes
  - Abstract Data Types
- A class can implement any number of interfaces
  - It must have concrete methods for the operations
- You can declare the type of a variable to be an interface
  - This is just like declaring the type to be an abstract class
- Important interfaces in Java's library include
  - `Runnable`, `Collection`, `Iterator`, `Comparable`, `Cloneable`

Packages and importing

- A package combines related classes into subsystems
  - All the classes in a particular directory
- Classes in different packages can have the same name
  - Although not recommended
- *Importing* a package is done as follows:

```
import finance.banking.accounts.*;
```


Access control

- Applies to methods and variables
  - `public`
    - —Any class can access
  - `protected`
    - —Only code in the package, or subclasses can access
  - (blank)
    - —Only code in the package can access
  - `private`
    - —Only code written in the class can access
    - —Inheritance still occurs!

Threads and concurrency

- Thread:
  - Sequence of executing statements that can be running concurrently with other threads
- To create a thread in Java:
  1. Create a class implementing `Runnable` or extending `Thread`
  2. Implement the `run` method as a loop that does something for a period of time
  3. Create an instance of this class
  4. Invoke the `start` operation, which calls `run`

Programming Style Guidelines

Remember that programs are for people to read
- Always choose the simpler alternative
- Reject clever code that is hard to understand
- Shorter code is not necessarily better

Choose good names
- Make them highly descriptive
- Do not worry about using long names

Programming style …

Comment extensively
- Comment whatever is non-obvious
- Do not comment the obvious
- Comments should be 25-50% of the code

Organize class elements consistently
- Variables, constructors, public methods then private methods

Be consistent regarding layout of code

Programming style …

Avoid duplication of code
- Do not 'clone' if possible
  - —Create a new method and call it
  - —Cloning results in two copies that may both have bugs
    - When one copy of the bug is fixed, the other may be forgotten

Programming style ...

Adhere to good object oriented principles
- E.g. the 'isa rule'

Prefer **private** as opposed to **public**

Do not mix user interface code with non-user interface code
- Interact with the user in separate classes
  - —This makes non-UI classes more reusable

2.10 Difficulties and Risks in Object-Oriented Programming

Language evolution and deprecated features:

- Java can be less efficient than other languages
  - —VM-based
  - —Dynamic binding

Efficiency can be a concern in some object oriented systems

- Java is evolving, so some features are ‘deprecated’ at every release
- But the same thing is true of most other languages

Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapter 3:

Basing Software Development on Reusable Technology

3.1 Building on the Experience of Others

Software engineers should avoid re-developing software already developed

Types of reuse:

- Reuse of expertise
- Reuse of standard designs and algorithms
- Reuse of libraries of classes or procedures
- Reuse of powerful commands built into languages and operating systems
- Reuse of frameworks
- Reuse of complete applications

3.2 Reusability and Reuse in SE

Reuse and design for *reusability* should be part of the culture of software development organizations

But there are problems to overcome:

- Why take the extra time needed to develop something that will benefit *other* projects/customers?
- Management may only reward the efforts of people who create the visible 'final products'.
- Reusable software are often created in a hurry and without enough attention to quality.

A vicious cycle

Developers tend not develop high quality reusable components, so there is often little to reuse

To solve the problem, recognize that:

- This vicious cycle costs money
- *Investment* in reusable code is important
- Attention to *quality* of reusable components is essential
  - —So that potential reusers have confidence in them
  - —The quality of a software product is only as good as its lowest-quality reusable component
- Developing reusable components can often simplify design

3.3 Frameworks: Reusable Subsystems

A *framework* is reusable software that implements a generic solution to a generalized problem.

It provides common facilities applicable to different application programs.

Principle: Applications that do different, but related, things tend to have quite similar designs

Frameworks to promote reuse

A framework is intrinsically *incomplete*

Certain classes or methods are used by the framework, but are missing (*slots*)

Some functionality is optional

— Allowance is made for developer to provide it (*hooks*)

Developers use the *services* that the framework provides

—Taken together the services are called the Application Program Interface (*API*)

Object-oriented frameworks

In the object oriented paradigm, a framework is composed of a library of classes.

The API is defined by the set of all public methods of these classes.

Some of the classes will normally be abstract

Examples of frameworks

A framework for payroll management

A framework for frequent buyer clubs

A framework for university registration

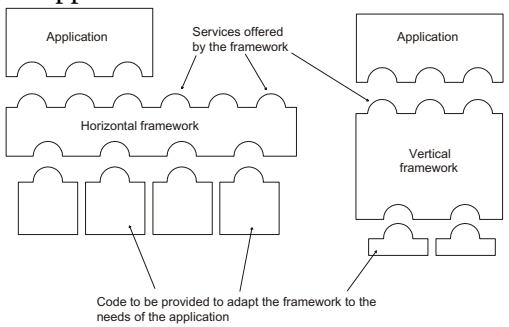
A framework for e-commerce web sites

A framework for controlling microwave ovens

Types of frameworks

A *horizontal* framework provides general application facilities that a large number of applications can use

A *vertical* framework (*application framework)* is more 'complete' but still needs some slots to be filled to adapt it to specific application needs



3.4 The Client-Server Architecture

A *distributed system* is a system in which:

computations are performed by *separate programs*

… normally running on separate pieces of hardware

… that *co-operate* to perform the task of the system.

Server:

A program that provides a service for other programs that connect to it using a communication channel

Client

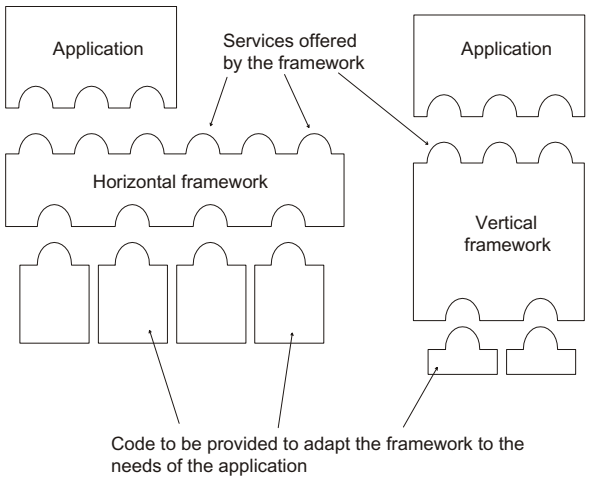
A program that accesses a server (or several servers) to obtain services

A server may be accessed by many clients simultaneously

Sequence of activities in a client-server system

1. The server starts running
2. The server waits for clients to connect. (*listening*)
3. Clients start running and perform operations
   — Some operations involve requests to the server
4. When a client attempts to connect, the server accepts the connection (if it is willing)
5. The server waits for messages to arrive from connected clients
6. When a message from a client arrives, the server takes some action in response, then resumes waiting
7. Clients and servers continue functioning in this manner until they decide to shut down or disconnect

A server program communicating with two client programs



Alternatives to the client server architecture

Have a *single program* on one computer that does everything

Have *no communication*

— Each computer performs the work separately

Have some mechanism other than client-server communication for exchanging information

—E.g. one program writes to a database; the other reads from the database

Advantages of client-server systems

The work can be *distributed* among different machines

The clients can access the server's functionality from a *distance*

The client and server can be *designed separately*

They can both be *simpler*

All the *data can be kept centrally* at the server

Conversely, *data can be distributed* among many different geographically-distributed clients or servers

The server can be accessed *simultaneously* by many clients

Competing clients can be written to communicate with the same server, and vice-versa

Example of client-server systems

The World Wide Web

Email

Network File System

Transaction Processing System

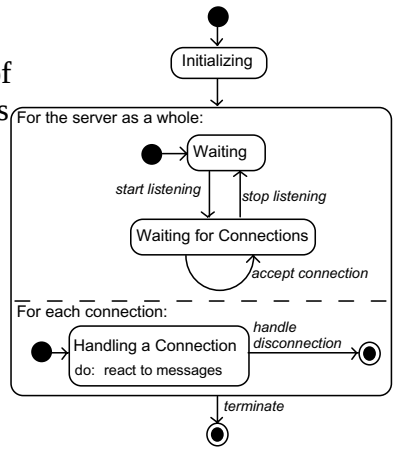
Remote Display System

Communication System

Database System

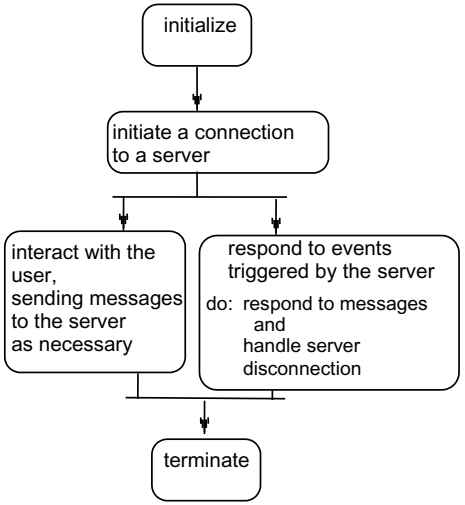
Activities of a server

1. Initializes itself
2. Starts listening for clients
3. Handles the following types of events originating from clients
   1. accepts connections
   2. responds to messages
   3. handles client disconnection
4. May stop listening
1. Must cleanly terminate

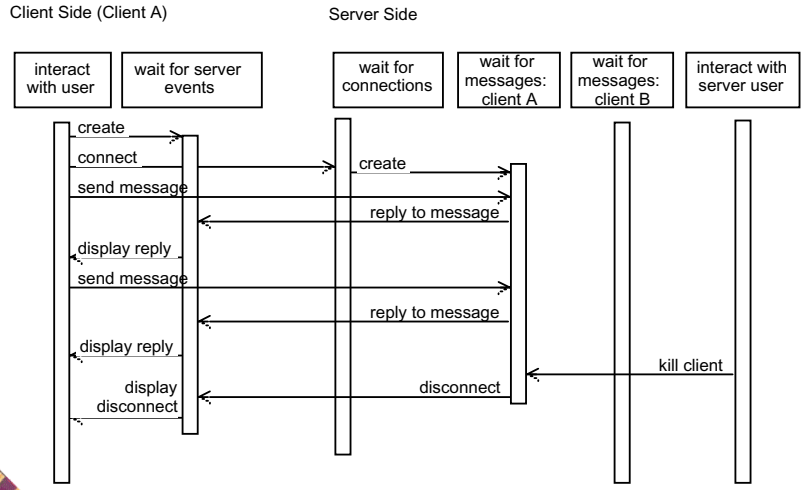


Activities of a client

1. Initializes itself
2. Initiates a connection
3. Sends messages
4. Handles the following types of events originating from the server
   1. responds to messages
   2. handles server disconnection
5. Must cleanly terminate



Threads in a client-server system



Thin- versus fat-client systems

Thin-client system (a)

Client is made as small as possible

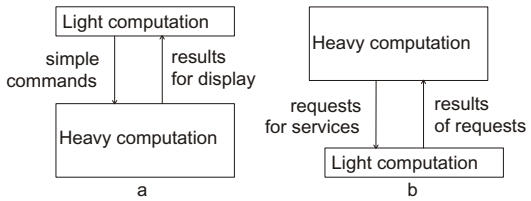
Most of the work is done in the server.

Client easy to download over the network

Fat-client system (b)

As much work as possible is delegated to the clients.

Server can handle more clients



Communications protocols

The messages the client sends to the server form a *language*.

— The server has to be programmed to understand that language.

The messages the server sends to the client also form a language.

— The client has to be programmed to understand that language.

When a client and server are communicating, they are in effect having a conversation using these two languages

The two languages and the rules of the conversation, taken together, are called the *protocol*

Tasks to perform to develop client-server applications

1. Design the primary work to be performed by both client and server
2. Design how the work will be distributed
3. Design the details of the set of messages that will be sent
4. Design the mechanism for
   1. Initializing
   2. Handling connections
   3. Sending and receiving messages
   4. Terminating

3.5 Technology Needed to Build Client-Server Systems

Internet Protocol (IP)
- Route messages from one computer to another
- Long messages are normally split up into small pieces

Transmission Control Protocol (TCP)
- Handles *connections* between two computers
- Computers can then exchange many IP messages over a connection
- Assures that the messages have been satisfactorily received

A host has an *IP address* and a *host name*
- Several servers can run on the same host.
- Each server is identified by a port number (0 to 65535).
- To initiate communication with a server, a client must know both the host name and the port number

Establishing a connection in Java

The **java.net** package
- Permits the creation of a TCP/IP connection between two applications

Before a connection can be established, the server must start *listening* to one of the ports:

```
ServerSocket serverSocket = new
    ServerSocket(port);
Socket clientSocket = serverSocket.accept();
```

For a client to connect to a server:

```
Socket clientSocket= new Socket(host, port);
```


Exchanging information in Java

Each program uses an instance of
- —`InputStream` to receive messages from the other program
- — `OutputStream` to send messages to the other program
- —These are found in package **`java.io`**

```
output = new
  OutputStream(clientSocket.getOutputStream());

input = new
  InputStream(clientSocket.getInputStream());
```

Sending and receiving messages

without any filters

```
output.write(msg);
msg = input.read();
```

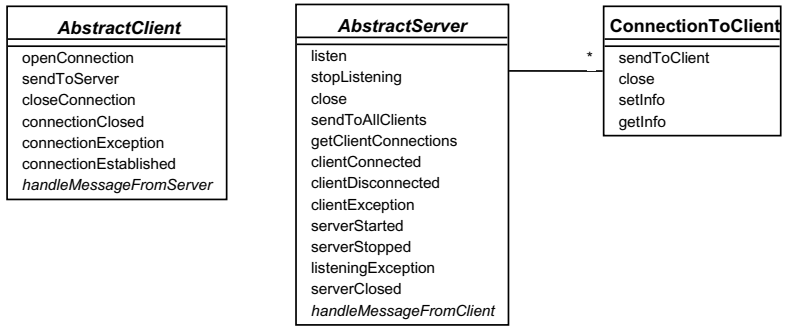
or using **DataInputStream** / **DataInputStream** filters

```
output.writeDouble(msg);
msg = input.readDouble();
```

or using **ObjectInputStream** / **ObjectInputStream** filters

```
output.writeObject(msg);
msg = input.readObject();
```


3.6 The Object Client-Server Framework (OCSF)



Using OCSF

Software engineers using OCSF *never* modify its three classes

They:

Create subclasses of the abstract classes in the framework

Call public methods that are provided by the framework

Override certain slot and hook methods (explicitly designed to be overridden)

3.7 The Client Side

Consists of a single class: `AbstractClient`

Must be subclassed

—Any subclass must provide an implementation for `handleMessageFromServer`

- Takes appropriate action when a message is received from a server

Implements the `Runnable` interface

—Has a `run` method which

- Contains a loop that executes for the lifetime of the thread

The public interface of AbstractClient

Controlling methods:
- **openConnection**
- **closeConnection**
- **sendToServer**

Accessing methods:
- **isConnected**
- **getHost**
- **setHost**
- **getPort**
- **setPort**
- **getInetAddress**

The callback methods of AbstractClient

Methods that *may* be overridden:
- **connectionEstablished**
- **connectionClosed**

Method that *must* be overridden:
- **handleMessageFromServer**

Using AbstractClient

Create a subclass of **AbstractClient**

Implement **handleMessageFromServer** slot method

Write code that:
- —Creates an instance of the new subclass
- —Calls **openConnection**
- —Sends messages to the server using the **sendToServer** service method

Implement the **connectionClosed** callback

Implement the **connectionException** callback

Internals of AbstractClient

Instance variables:

A **Socket** which keeps all the information about the connection to the server

Two streams, an **ObjectOutputStream** and an **ObjectInputStream**

A **Thread** that runs using **AbstractClient**'s run method

Two variables storing the *host* and *port* of the server

3.8 The Server Side

Two classes:

One for the thread which listens for new connections (**`AbstractServer`**)

One for the threads that handle the connections to clients (**`ConnectionToClient`**)

The public interface of `AbstractServer`

Controlling methods:

- **`listen`**
- **`stopListening`**
- **`close`**
- **`sendToAllClients`**

Accessing methods:

- **`isListening`**
- **`getClientConnections`**
- **`getPort`**
- **`setPort`**
- **`setBacklog`**

The callback methods of `AbstractServer`

Methods that *may* be overridden:

- **`serverStarted`**
- **`clientConnected`**
- **`clientDisconnected`**
- **`clientException`**
- **`serverStopped`**
- **`listeningException`**
- **`serverClosed`**

Method that *must* be overridden:

- **`handleMessageFromClient`**

The public interface of `ConnectionToClient`

Controlling methods:

- **`sendToClient`**
- **`close`**

Accessing methods:

- **`getInetAddress`**
- **`setInfo`**
- **`getInfo`**

Using AbstractServer and ConnectionToClient

Create a subclass of **AbstractServer**

Implement the slot method **handleMessageFromClient**

Write code that:

— Creates an instance of the subclass of **AbstractClient**

— Calls the **listen** method

— Sends messages to clients, using:

- the **getClientConnections** and **sendToClient** service methods
- or **sendToAllClients**

Implement one or more of the other callback methods

Internals of AbstractServer and ConnectionToClient

The **setInfo** and **getInfo** methods make use of a Java class called **HashMap**

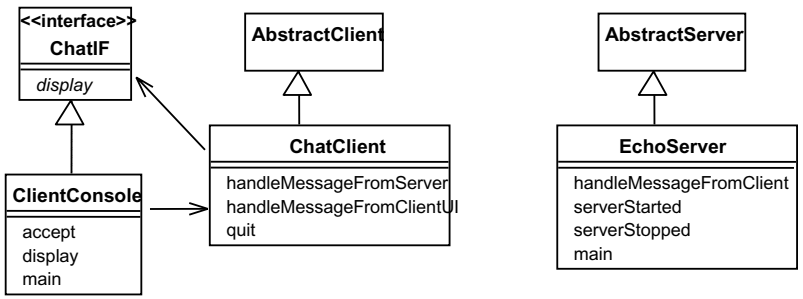
Many methods in the server side are **synchronized**

The collection of instances of **ConnectionToClient** is stored using a special class called **ThreadGroup**

The server must pause from listening every 500ms to see if the **stopListening** method has been called

—if not, then it resumes listening immediately

3.11 An Instant Messaging Application: SimpleChat



ClientConsole can eventually be replaced by ClientGUI

The server

EchoServer is a subclass of **AbstractServer**

The **main** method creates a new instance and starts it

— It listens for clients and handles connections until the server is stopped

The three *callback* methods just print out a message to the user

— **handleMessageFromClient**, **serverStarted** and **serverStopped**

The *slot* method **handleMessageFromClient** calls **sendToAllClients**

— This echoes any messages

Key code in EchoServer

```
public void handleMessageFromClient
  (Object msg, ConnectionToClient client)
{
   System.out.println(
     "Message received: "
     + msg + " from " + client);
   this.sendToAllClients(msg);
}
```


The client

When the client program starts, it creates instances of two classes:

- **ChatClient**
  - —A subclass of **AbstractClient**
  - —Overrides **handleMessageFromServer**
    - This calls the **display** method of the user interface
- **ClientConsole**
  - —User interface class that implements the interface **ChatIF**
    - Hence implements **display** which outputs to the console
  - —Accepts user input by calling **accept** in its **run** method
  - —Sends all user input to the **ChatClient** by calling its **handleMessageFromClientUI**
    - This, in turn, calls **sendToServer**

Key code in ChatClient

```
public void handleMessageFromClientUI(
  String message)
{
  try
  {
    sendToServer(message);
  }
  catch(IOException e)
  {
    clientUI.display (
      "Could not send message. " +
      "Terminating client.");
    quit();
  }
}
```


Key code in ChatClient - continued

```
public void handleMessageFromServer(Object msg)
{
  clientUI.display(msg.toString());
}
```

3.12 Risks when reusing technology

Poor quality reusable components

—*Ensure that the developers of the reusable technology:*

- *follow good software engineering practices*
- *are willing to provide active support*

Compatibility not maintained

—*Avoid obscure features*

—*Only re-use technology that others are also re-using*

Risks when developing reusable technology

Investment uncertainty

—*Plan the development of the reusable technology, just as if it was a product for a client*

The 'not invented here syndrome'

—*Build confidence in the reusable technology by:*

- *Guaranteeing support*
- *Ensuring it is of high quality*
- *Responding to the needs of its users*

Risk when developing reusable technology – continued

Competition

—*The reusable technology must be as useful and as high quality as possible*

Divergence (tendency of various groups to change technology in different ways)

—*Design it to be general enough, test it and review it in advance*

Risks when adopting a client-server approach

Security

—*Security is a big problem with no perfect solutions: consider the use of encryption, firewalls, ...*

Need for adaptive maintenance

—*Ensure that all software is forward and backward compatible with other versions of clients and servers*

Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapter 4:

Developing Requirements

4.1 Domain Analysis

The process by which a software engineer learns about the domain to better understand the problem:

- The *domain* is the general field of business or technology in which the clients will use the software
- A *domain expert* is a person who has a deep knowledge of the domain

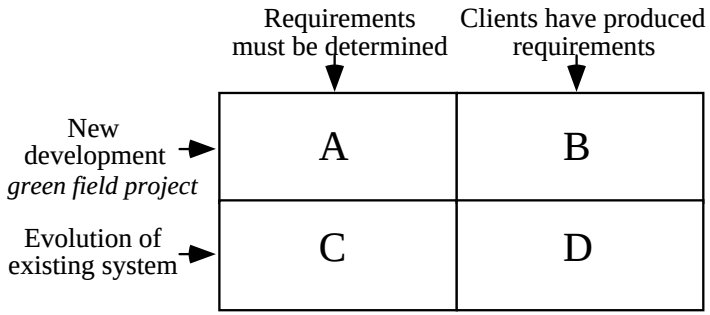
Benefits of performing domain analysis:

- Faster development
- Better system
- Anticipation of extensions

Domain Analysis document

A. Introduction
B. Glossary
C. General knowledge about the domain
D. Customers and users
E. The environment
F. Tasks and procedures currently performed
G. Competing software
H. Similarities to other domains

4.2 The Starting Point for Software Projects



4.3 Defining the Problem and the Scope

A problem can be expressed as:

- A *difficulty* the users or customers are facing,
- Or as an *opportunity* that will result in some benefit such as improved productivity or sales.

The solution to the problem normally will entail developing software

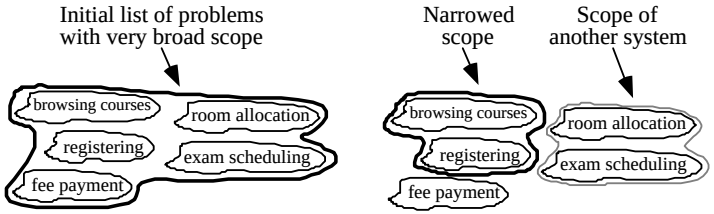
A good problem statement is short and succinct

Defining the Scope

Narrow the *scope* by defining a more precise problem

- List all the things you might imagine the system doing
  - —Exclude some of these things if too broad
  - —Determine high-level goals if too narrow

Example: A university registration system



4.4 What is a Requirement

Requirement: A statement about the proposed system that all stakeholders agree must be made true in order for the customer's problem to be adequately solved.

- Short and concise piece of information
- Says something about the system
- All the stakeholders have agreed that it is valid
- It helps solve the customer's problem

A collection of requirements is a *requirements document.*

4.5 Types of Requirements

Functional requirements

- Describe *what* the system should do

Non-functional requirements

- C*onstraints* that must be adhered to during development

Functional requirements

What *inputs* the system should accept

What *outputs* the system should produce

What data the system should *store* that other systems might use

What *computations* the system should perform

The *timing and synchronization* of the above

Non-functional requirements

All must be verifiable

Three main types

1. Categories reflecting: usability, efficiency, reliability, maintainability and reusability
   - —Response time
   - —Throughput
   - —Resource usage
   - —Reliability
   - —Availability
   - —Recovery from failure
   - —Allowances for maintainability and enhancement
   - —Allowances for reusability

Non-functional requirements

2. Categories constraining the *environment and technology* of the system.
   - —Platform
   - —Technology to be used

3. Categories constraining the *project plan and development methods*
   - —Development process (methodology) to be used
   - —Cost and delivery date
     - Often put in contract or project plan instead

4.6 Some Techniques for Gathering and Analysing Requirements

Observation

- Read documents and discuss requirements with users
- Shadowing important potential users as they do their work
  - —ask the user to explain everything he or she is doing
- Session videotaping

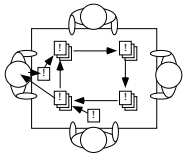
Interviewing

- Conduct a series of interviews
  - —Ask about specific details
  - —Ask about the stakeholder's vision for the future
  - —Ask if they have alternative ideas
  - —Ask for other sources of information
  - —Ask them to draw diagrams

Gathering and Analysing Requirements...

Brainstorming

- Appoint an experienced moderator
- Arrange the attendees around a table
- Decide on a 'trigger question'
- Ask each participant to write an answer and pass the paper to its neighbour



Joint Application Development (JAD) is a technique based on intensive brainstorming sessions

Gathering and Analysing Requirements...

Prototyping

- The simplest kind: *paper prototype.*
  - —a set of pictures of the system that are shown to users in sequence to explain what would happen
- The most common: a mock-up of the system's UI
  - —Written in a rapid prototyping language
  - —Does *not* normally perform any computations, access any databases or interact with any other systems
  - —May prototype a particular aspect of the system

Gathering and Analysing Requirements...

Informal use case analysis

- Determine the classes of users that will use the facilities of this system (actors)
- Determine the tasks that each actor will need to do with the system

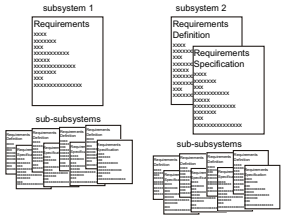
More on use cases in Chapter 7

4.7 Types of Requirements Document

Two extremes:

- An informal outline of the requirements using a few paragraphs or simple diagrams
  - requirements *definition*
- A long list of specifications that contain thousands of pages of intricate detail
  - requirements *specification*

Requirements documents for large systems are normally arranged in a hierarchy



Level of detail required in a requirements document

How much detail should be provided depends on:

- —The size of the system
- —The need to interface to other systems
- —The readership
- —The stage in requirements gathering
- —The level of experience with the domain and the technology
- —The cost that would be incurred if the requirements were faulty

4.8 Reviewing Requirements

Each individual requirement should

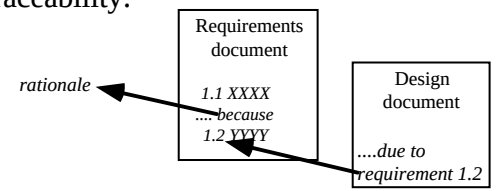
- —Have benefits that outweigh the costs of development
- —Be important for the solution of the current problem
- —Be expressed using a clear and consistent notation
- —Be unambiguous
- —Be logically consistent
- —Lead to a system of sufficient quality
- —Be realistic with available resources
- —Be verifiable
- —Be uniquely identifiable
- —Does not over-constrain the design of the system

Requirements documents...

The document should be:

- —sufficiently complete
- —well organized
- —clear
- —agreed to by all the stakeholders

Traceability:



Requirements document...

A. Problem
B. Background information
C. Environment and system models
D. Functional Requirements
E. Non-functional requirements

4.9 Managing Changing Requirements

Requirements change because:

- Business process changes
- Technology changes
- The problem becomes better understood

Requirements analysis never stops

- Continue to interact with the clients and users
- The benefits of changes must outweigh the costs.
  - —Certain small changes (e.g. look and feel of the UI) are usually quick and easy to make at relatively little cost.
  - —Larger-scale changes have to be carefully assessed
    - Forcing unexpected changes into a partially built system will probably result in a poor design and late delivery
- Some changes are enhancements in disguise
  - —Avoid making the system *bigger*, only make it *better*

4.13 Difficulties and Risks in Domain and Requirements Analysis

Lack of understanding of the domain or the real problem

- *—Do domain analysis and prototyping*

Requirements change rapidly

- *—Perform incremental development, build flexibility into the design, do regular reviews*

Attempting to do too much

- *—Document the problem boundaries at an early stage, carefully estimate the time*

It may be hard to reconcile conflicting sets of requirements

- *—Brainstorming, JAD sessions, competing prototypes*

It is hard to state requirements precisely

- *—Break requirements down into simple sentences and review them carefully, look for potential ambiguity, make early prototypes*

Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapter 5:

Modelling with Classes

5.1 What is UML?

The Unified Modelling Language is a standard graphical language for modelling object oriented software

- At the end of the 1980s and the beginning of 1990s, the first object-oriented development processes appeared
- The proliferation of methods and notations tended to cause considerable confusion
- Two important methodologists Rumbaugh and Booch decided to merge their approaches in 1994.
  - —They worked together at the Rational Software Corporation
- In 1995, another methodologist, Jacobson, joined the team
  - —His work focused on use cases
- In 1997 the Object Management Group (OMG) started the process of UML standardization

UML diagrams

- Class diagrams
  - —describe classes and their relationships
- Interaction diagrams
  - —show the behaviour of systems in terms of how objects interact with each other
- State diagrams and activity diagrams
  - —show how systems behave internally
- Component and deployment diagrams
  - —show how the various components of systems are arranged logically and physically

UML features

- It has detailed *semantics*
- It has *extension* mechanisms
- It has an associated textual language
  - —*Object Constraint Language* (OCL)

The objective of UML is to assist in software development

- —It is not a *methodology*

What constitutes a good model?

A model should
- use a standard notation
- be understandable by clients and users
- lead software engineers to have insights about the system
- provide abstraction

Models are used:
- to help create designs
- to permit analysis and review of those designs.
- as the core documentation describing the system.

5.2 Essentials of UML Class Diagrams

The main symbols shown on class diagrams are:

Classes
- represent the types of data themselves

Associations
- represent linkages between instances of classes

Attributes
- are simple data found in classes and their instances

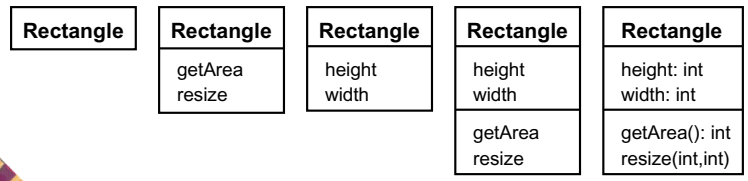
Operations
- represent the functions performed by the classes and their instances

Generalizations
- group classes into inheritance hierarchies

Classes

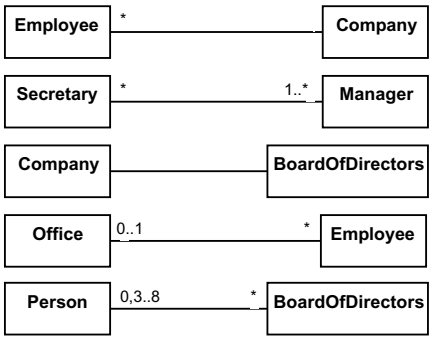
A class is simply represented as a box with the name of the class inside
- The diagram may also show the attributes and operations
- The complete signature of an operation is:

operationName(parameterName: parameterType …): returnType



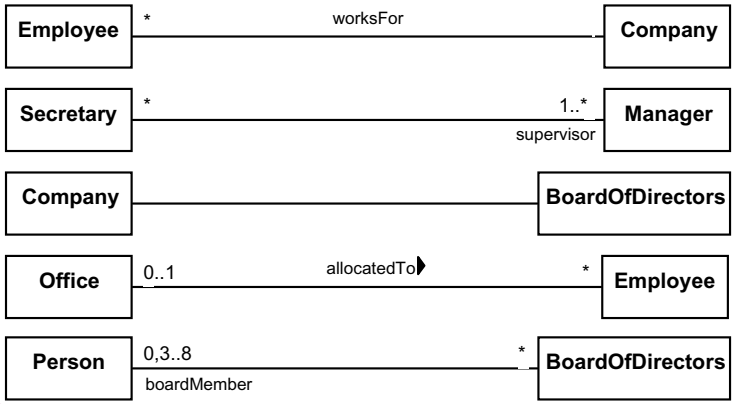
5.3 Associations and Multiplicity

An *association* is used to show how two classes are related to each other
- Symbols indicating *multiplicity* are shown at each end of the association



Labelling associations

Each association can be labelled, to make explicit the nature of the association



Analyzing and validating associations

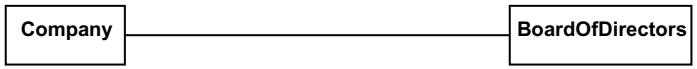
One-to-one

—For each company, there is exactly one board of directors

—A board is the board of only one company

—A company must always have a board

—A board must always be of some company



Analyzing and validating associations

Many-to-many

—A secretary can work for many managers

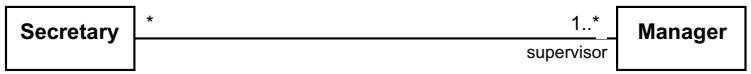
—A manager can have many secretaries

—Secretaries can work in pools

—Managers can have a group of secretaries

—Some managers might have zero secretaries.

—Is it possible for a secretary to have, perhaps temporarily, zero managers?



Analyzing and validating associations

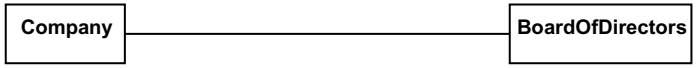
One-to-one

—For each company, there is exactly one board of directors

—A board is the board of only one company

—A company must always have a board

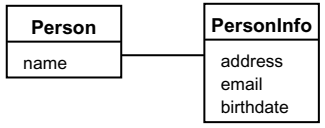
—A board must always be of some company



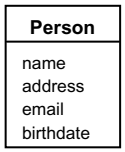
Analyzing and validating associations

Avoid unnecessary one-to-one associations

Avoid this



do this



A more complex example

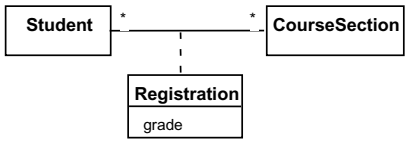
A booking is always for exactly one passenger
- —no booking with zero passengers
- —a booking could *never* involve more than one passenger.

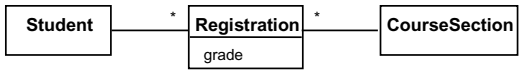
A Passenger can have any number of Bookings
- —a passenger could have no bookings at all
- —a passenger could have more than one booking

Association classes

Sometimes, an attribute that concerns two associated classes cannot be placed in either of the classes

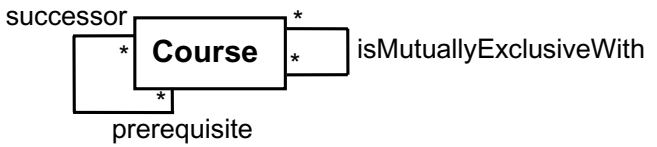
The following are equivalent





Reflexive associations

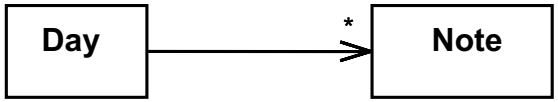
It is possible for an association to connect a class to itself



Directionality in associations

Associations are by default *bi-directional*

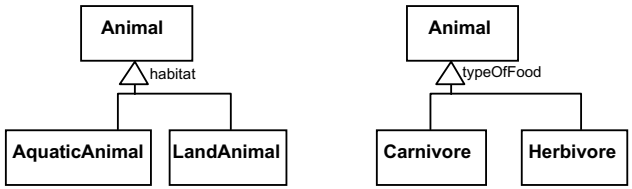
It is possible to limit the direction of an association by adding an arrow at one end



5.4 Generalization

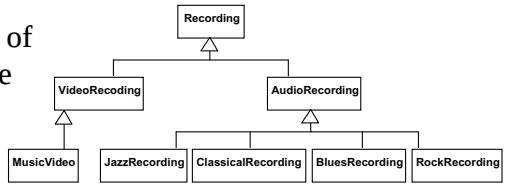
Specializing a superclass into two or more subclasses

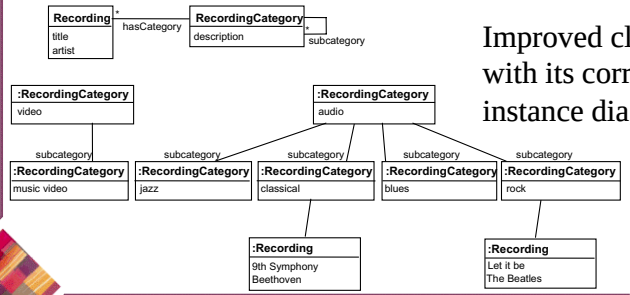
The *discriminator* is a label that describes the criteria used in the specialization



Avoiding unnecessary generalizations

Inappropriate hierarchy of classes, which should be instances

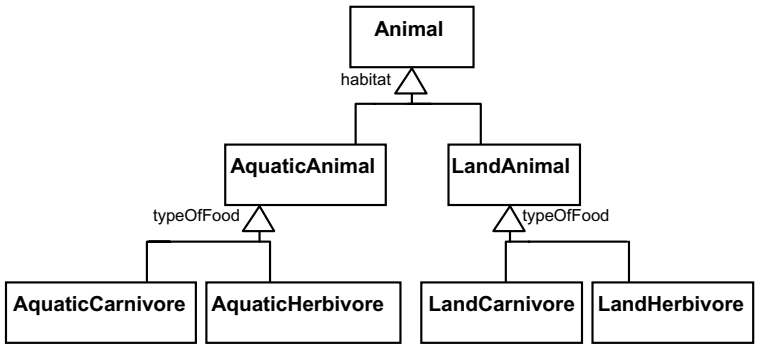




Improved class diagram, with its corresponding instance diagram

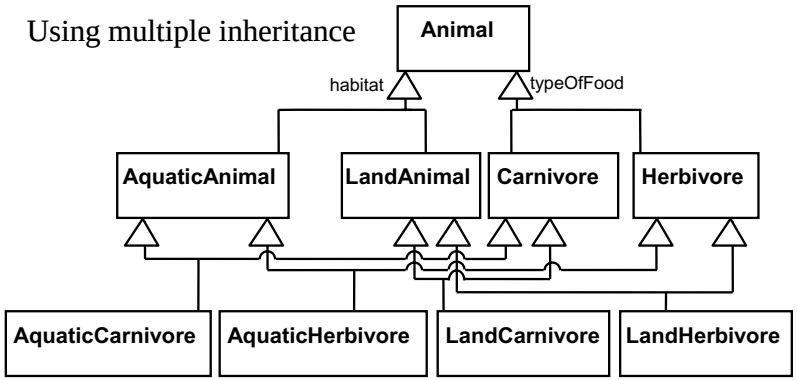
Handling multiple discriminators

Creating higher-level generalization



Handling multiple discriminators

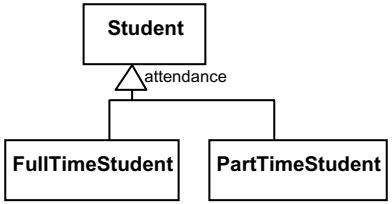
Using multiple inheritance



Using the Player-Role pattern (in Chapter 6)

Avoiding having instances change class

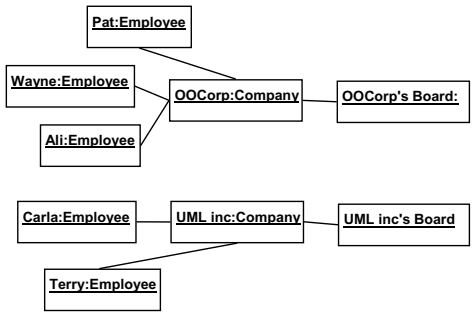
An instance should never need to change class



5.5 Instance Diagrams

A *link* is an instance of an association

—In the same way that we say an object is an instance of a class



Associations versus generalizations in instance diagrams

Associations describe the relationships that will exist between *instances* at run time.

—When you show an instance diagram generated from a class diagram, there will be an instance of *both* classes joined by an association

Generalizations describe relationships between *classes* in class diagrams.

—They do not appear in instance diagrams at all.

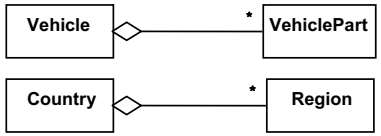
—An instance of any class should also be considered to be an instance of each of that class's superclasses

5.6 More Advanced Features: Aggregation

Aggregations are special associations that represent 'part-whole' relationships.

—The 'whole' side is often called the *assembly* or the *aggregate*

—This symbol is a shorthand notation association named `isPartOf`



When to use an aggregation

As a general rule, you can mark an association as an aggregation if the following are true:

You can state that

—the parts 'are part of' the aggregate

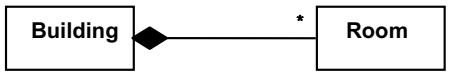
—or the aggregate 'is composed of' the parts

When something owns or controls the aggregate, then they also own or control the parts

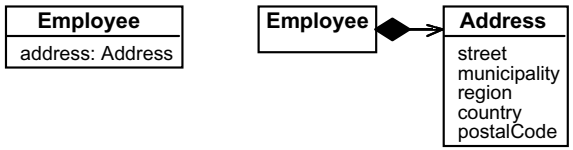
Composition

A *composition* is a strong kind of aggregation

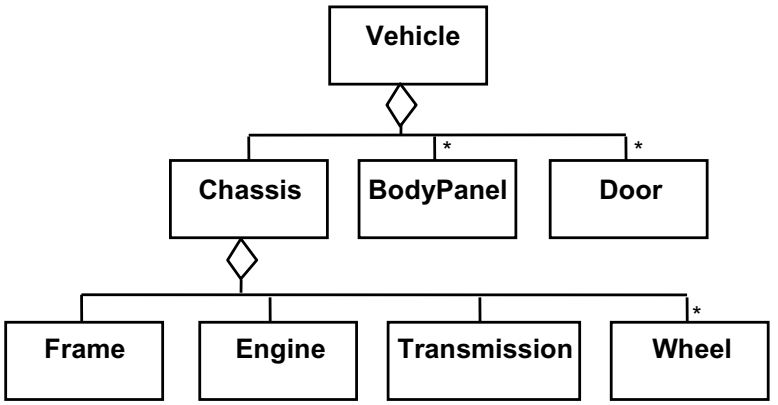
—if the aggregate is destroyed, then the parts are destroyed as well



Two alternatives for addresses



Aggregation hierarchy



Propagation

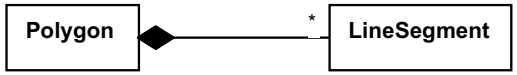
A mechanism where an operation in an aggregate is implemented by having the aggregate perform that operation on its parts

At the same time, properties of the parts are often propagated back to the aggregate

Propagation is to aggregation as inheritance is to generalization.

—The major difference is:

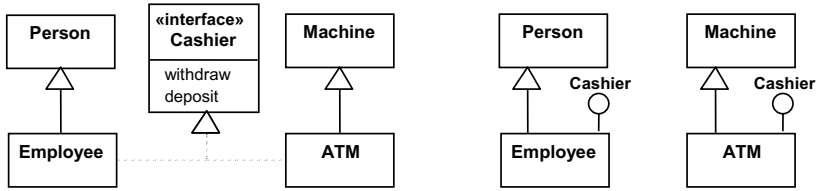
- inheritance is an implicit mechanism
- propagation has to be programmed when required



Interfaces

An interface describes a *portion of the visible behaviour* of a set of objects.

An *interface* is similar to a class, except it lacks instance variables and implemented methods



Notes and descriptive text

Descriptive text and other diagrams

—Embed your diagrams in a larger document

—Text can explain aspects of the system using any notation you like

—Highlight and expand on important features, and give rationale

Notes:

—A note is a small block of text embedded *in* a UML diagram

—It acts like a comment in a programming language

Object Constraint Language (OCL)

OCL is a *specification* language designed to formally specify constraints in software modules

An OCL expression simply specifies a logical fact (a constraint) about the system that must remain true

A constraint cannot have any side-effects

—it cannot compute a non-Boolean result nor modify any data.

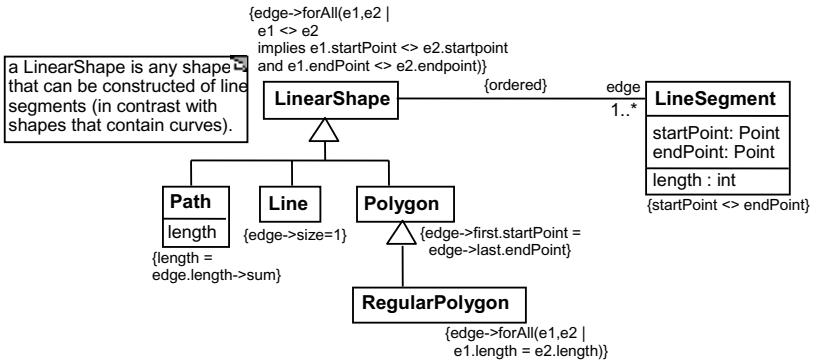
OCL statements in class diagrams can specify what the values of attributes and associations must be

OCL statements

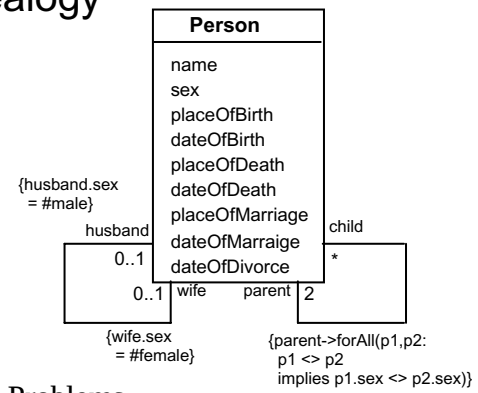
OCL statements can be built from:

- References to role names, association names, attributes and the results of operations
- The logical values **true** and **false**
- Logical operators such as **and**, **or**, **=**, **>**, **<** or **<>** (not equals)
- String values such as: **'a string'**
- Integers and real numbers
- Arithmetic operations *****, **/**, **+**, **-**

An example: constraints on Polygons



5.7 Detailed Example: A Class Diagram for Genealogy

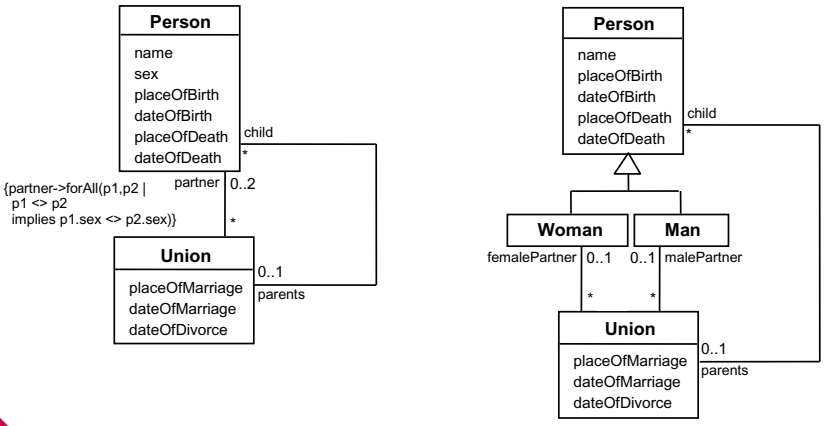


Problems

—A person must have two parents

—Marriages not properly accounted for

Genealogy example: Possible solutions



5.8 The Process of Developing Class Diagrams

You can create UML models at different stages and with different purposes and levels of details

Exploratory domain model:

—Developed in domain analysis to learn about the domain

System domain model:

—Models aspects of the domain represented by the system

System model:

—Includes also classes used to build the user interface and system architecture

System domain model vs System model

The *system domain model* omits many classes that are needed to build a complete system

—Can contain less than half the classes of the system.

—Should be developed to be used independently of particular sets of

- user interface classes
- architectural classes

The complete *system model* includes

—The system domain model

—User interface classes

—Architectural classes

—Utility classes

Suggested sequence of activities

Identify a first set of candidate classes

Add associations and attributes

Find generalizations

List the main responsibilities of each class

Decide on specific operations

Iterate over the entire process until the model is satisfactory

—Add or delete classes, associations, attributes, generalizations, responsibilities or operations

—Identify interfaces

—Apply design patterns (Chapter 6)

Don't be too disorganized. Don't be too rigid either.

Identifying classes

When developing a domain model you tend to *discover* classes

When you work on the user interface or the system architecture, you tend to *invent* classes

—Needed to solve a particular design problem

—(Inventing may also occur when creating a domain model)

Reuse should always be a concern

—Frameworks

—System extensions

—Similar systems

A simple technique for discovering domain classes

Look at a source material such as a description of requirements

Extract the *nouns* and *noun phrases*

Eliminate nouns that:

—are redundant

—represent instances

—are vague or highly general

—not needed in the application

Pay attention to classes in a domain model that represent *types of users* or other actors

Identifying associations and attributes

Start with classes you think are most central and important

Decide on the clear and obvious data it must contain and its relationships to other classes.

Work outwards towards the classes that are less important.

Avoid adding many associations and attributes to a class

—A system is simpler if it manipulates less information

Tips about identifying and specifying valid associations

An association should exist if a class

- *possesses*
- *controls*
- *is connected to*
- *is related to*
- *is a part of*
- *has as parts*
- *is a member of*, or
- *has as members*

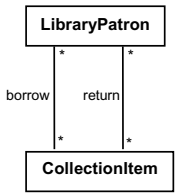
some other class in your model

Specify the multiplicity at both ends

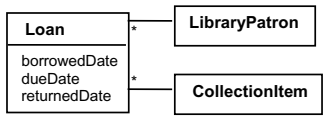
Label it clearly.

Actions versus associations

A common mistake is to represent *actions* as if they were associations



Bad, due to the use of associations that are actions



Better: The **`borrow`** operation creates a **`Loan`**, and the **`return`** operation sets the **`returnedDate`** attribute.

Identifying attributes

Look for information that must be maintained about each class

Several nouns rejected as classes, may now become attributes

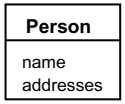
An attribute should generally contain a simple value

—E.g. string, number

Tips about identifying and specifying valid attributes

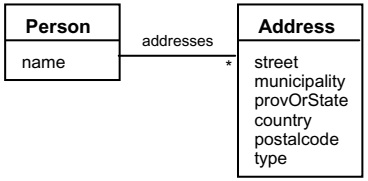
It is not good to have many duplicate attributes

If a subset of a class's attributes form a coherent group, then create a distinct class containing these attributes



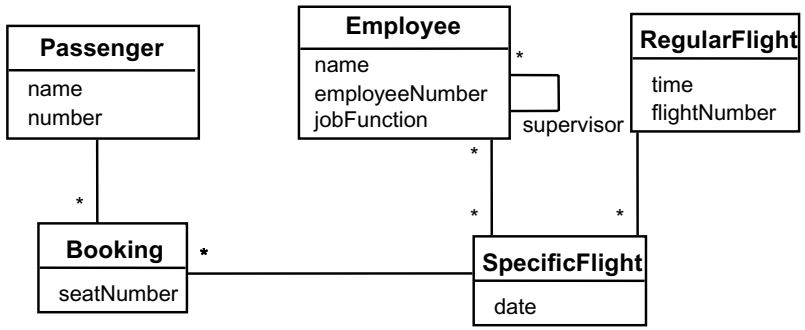
Bad due to a plural attribute

Bad due to too many attributes, and inability to add more addresses



Good solution. The type indicates whether it is a home address, business address etc.

An example (attributes and associations)



Identifying generalizations and interfaces

There are two ways to identify generalizations:

—bottom-up

- Group together similar classes creating a new superclass

—top-down

- Look for more general classes first, specialize them if needed

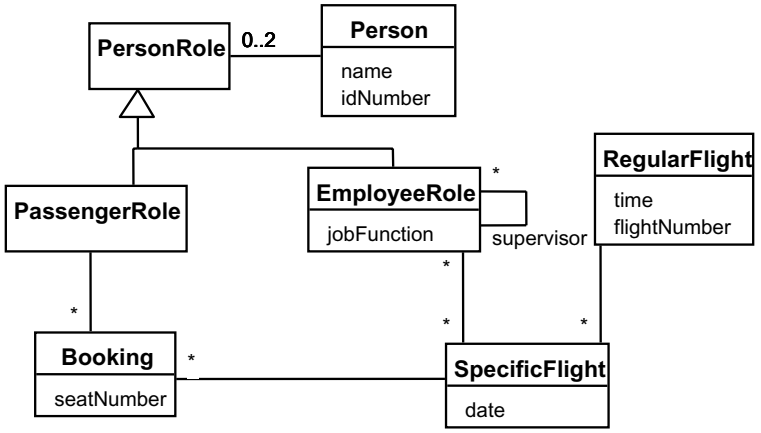
Create an *interface*, instead of a superclass if

—The classes are very dissimilar except for having a few operations in common

—One or more of the classes already have their own superclasses

—Different implementations of the same class might be available

An example (generalization)



Allocating responsibilities to classes

A *responsibility* is something that the system is required to do.

Each functional requirement must be attributed to one of the classes

—All the responsibilities of a given class should be *clearly related.*

—If a class has too many responsibilities, consider *splitting* it into distinct classes

—If a class has no responsibilities attached to it, then it is probably *useless*

—When a responsibility cannot be attributed to any of the existing classes, then a *new class* should be created

To determine responsibilities

—Perform use case analysis

—Look for verbs and nouns describing *actions* in the system description

Categories of responsibilities

Setting and getting the values of attributes

Creating and initializing new instances

Loading to and saving from persistent storage

Destroying instances

Adding and deleting links of associations

Copying, converting, transforming, transmitting or outputting

Computing numerical results

Navigating and searching

Other specialized work

An example (responsibilities)

—Creating a new regular flight

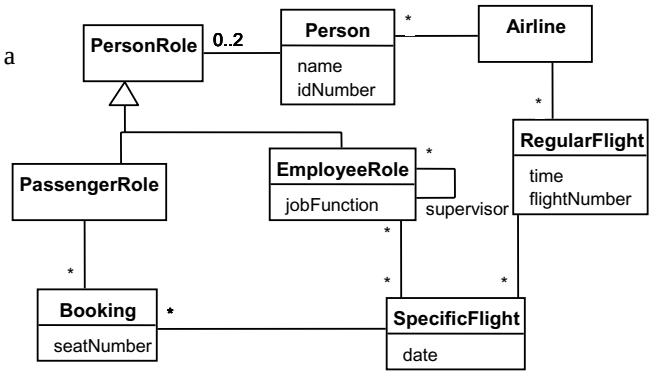
—Searching for a flight

—Modifying attributes of a flight

—Creating a specific flight

—Booking a passenger

—Canceling a booking



Prototyping a class diagram on paper

As you identify classes, you write their names on small cards

As you identify attributes and responsibilities, you list them on the cards

— If you cannot fit all the responsibilities on one card:

- this suggests you should split the class into two related classes.

Move the cards around on a whiteboard to arrange them into a class diagram.

Draw lines among the cards to represent associations and generalizations.

Identifying operations

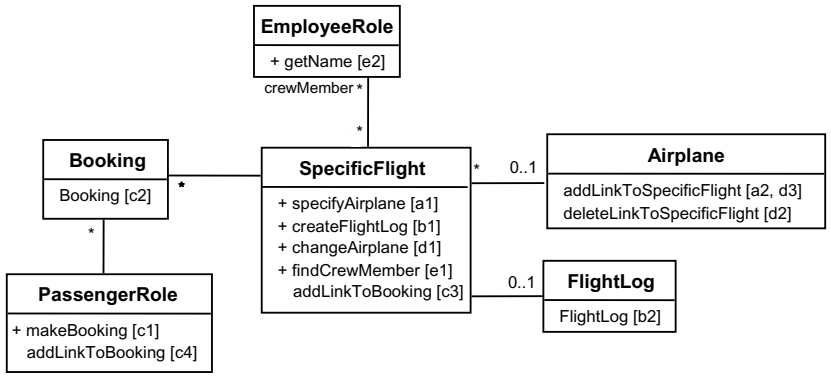
Operations are needed to realize the responsibilities of each class

There may be several operations per responsibility

The main operations that implement a responsibility are normally declared public

Other methods that collaborate to perform the responsibility must be as private as possible

An example (class collaboration)

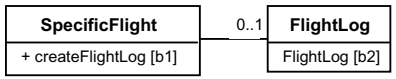


Class collaboration 'a'

Making a bi-directional link between two existing objects;
e.g. adding a link between an instance of **SpecificFlight** and an instance of **Airplane**.

1. (public) The instance of **SpecificFlight**
   — makes a one-directional link to the instance of Airplane
   — then calls operation 2.
2. (non-public) The instance of **Airplane**
   — makes a one-directional link back to the instance of SpecificFlight

Class collaboration ‘b’

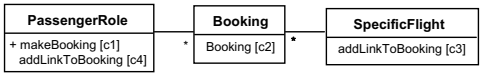


Creating an object and linking it to an existing object

e.g. creating a **FlightLog**, and linking it to a **SpecificFlight**.

1. (public) The instance of **SpecificFlight**
   - —calls the constructor of FlightLog (operation 2)
   - —then makes a one-directional link to the new instance of FlightLog.
2. (non-public) Class **FlightLog**’s constructor
   - —makes a one-directional link back to the instance of SpecificFlight.

Class collaboration ‘c’



Creating an association class, given two existing objects

e.g. creating an instance of **Booking**, which will link a **SpecificFlight** to a **PassengerRole**.

1. (public) The instance of **PassengerRole**
   - — calls the constructor of **Booking** (operation 2).
2. (non-public) Class **Booking**’s constructor, among its other actions
   - — makes a one-directional link back to the instance of **PassengerRole**
   - — makes a one-directional link to the instance of **SpecificFlight**
   - — calls operations 3 and 4.
3. (non-public) The instance of **SpecificFlight**
   - — makes a one-directional link to the instance of **Booking**.
4. (non-public) The instance of **PassengerRole**
   - — makes a one-directional link to the instance of **Booking.**

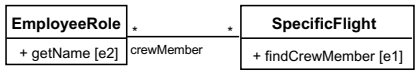
Class collaboration ‘d’

Changing the destination of a link

e.g. changing the **Airplane** of to a **SpecificFlight**, from **airplane1** to **airplane2**

1. (public) The instance of **SpecificFlight**
   - —deletes the link to **airplane1**
   - —makes a one-directional link to **airplane2**
   - —calls operation 2
   - —then calls operation 3.
2. (non-public) **airplane1**
   - —deletes its one-directional link to the instance of **SpecificFlight**.
3. (non-public) **airplane2**
   - —makes a one-directional link to the instance of **SpecificFlight**.

Class collaboration ‘e’



Searching for an associated instance

e.g. searching for a crew member associated with a **SpecificFlight** that has a certain name.

1. (public) The instance of **SpecificFlight**
   - — creates an Iterator over all the crewMember links of the SpecificFlight\
   - — for each of them call operation 2, until it finds a match.
2. (may be public) The instance of **EmployeeRole** returns its name.

5.9 Implementing Class Diagrams in Java

- Attributes are implemented as instance variables
- Generalizations are implemented using `extends`
- Interfaces are implemented using `implements`
- Associations are normally implemented using instance variables
  - Divide each two-way association into two one-way associations
    - —so each associated class has an instance variable.
  - For a one-way association where the multiplicity at the other end is 'one' or 'optional'
    - —declare a variable of that class (a reference)
  - For a one-way association where the multiplicity at the other end is 'many':
    - —use a collection class implementing `List`, such as `Vector`

Example: **SpecificFlight**

```
class SpecificFlight
{
  private Calendar date;
  private RegularFlight regularFlight;
  private TerminalOfAirport destination;
  private Airplane airplane;
  private FlightLog flightLog;

  private ArrayList crewMembers;
  // of EmployeeRole
  private ArrayList bookings
  ...
}
```


Example: **SpecificFlight**

```
// Constructor that should only be called from
// addSpecificFlight
SpecificFlight(
  Calendar aDate,
  RegularFlight aRegularFlight)
{
  date = aDate;
  regularFlight = aRegularFlight;
}
```


Example: **RegularFlight**

```
class RegularFlight
{
  private ArrayList specificFlights;
  ...
  // Method that has primary
  // responsibility

  public void addSpecificFlight(
    Calendar aDate)
  {
    SpecificFlight newSpecificFlight;
    newSpecificFlight =
      new SpecificFlight(aDate, this);
    specificFlights.add(newSpecificFlight);
  }
  ...
}
```

Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapter 6:

Using Design Patterns

6.1 Introduction to Patterns

The recurring aspects of designs are called *design patterns.*

A *pattern* is the outline of a reusable solution to a general problem encountered in a particular context

Many of them have been systematically documented for all software developers to use

A good pattern should

—Be as general as possible

—Contain a solution that has been proven to effectively solve the problem in the indicated context.

Studying patterns is an effective way to learn from the experience of others

Pattern description

Context:

The general situation in which the pattern applies

Problem:

—A short sentence or two raising the main difficulty.

Forces:

The issues or concerns to consider when solving the problem

Solution:

The recommended way to solve the problem in the given context.

—'to balance the forces'

Antipatterns: (Optional)

Solutions that are inferior or do not work in this context.

Related patterns: (Optional)

Patterns that are similar to this pattern.

References:

Who developed or inspired the pattern.

6.2 The Abstraction-Occurrence Pattern

Context:

—Often in a domain model you find a set of related objects (*occurrences*).

—The members of such a set share common information

- but also differ from each other in important ways.

Problem:

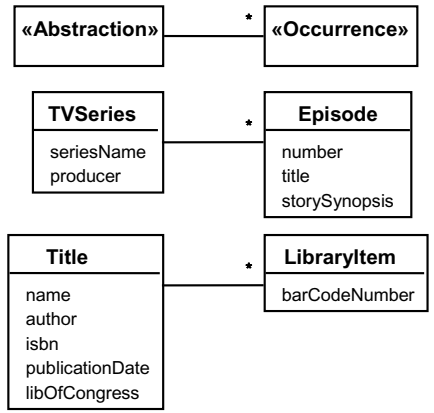
—What is the best way to represent such sets of occurrences in a class diagram?

Forces:

—You want to represent the members of each set of occurrences without duplicating the common information

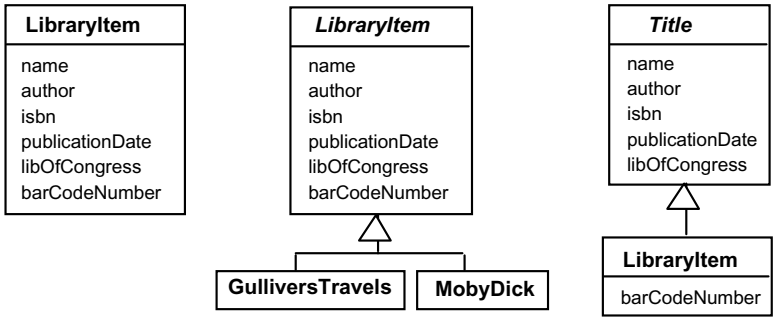
Abstraction-Occurrence

Solution:



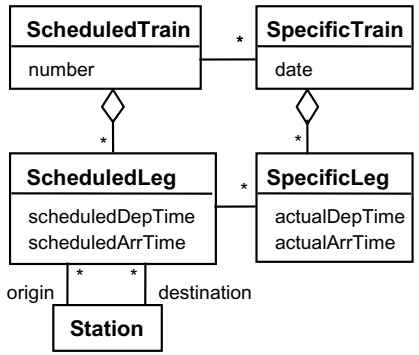
Abstraction-Occurrence

Antipatterns:



Abstraction-Occurrence

Square variant



6.3 The General Hierarchy Pattern

Context:

—Objects in a hierarchy can have one or more objects above them (superiors),

- and one or more objects below them (subordinates).

—Some objects cannot have any subordinates

Problem:

—How do you represent a hierarchy of objects, in which some objects cannot have subordinates?

Forces:

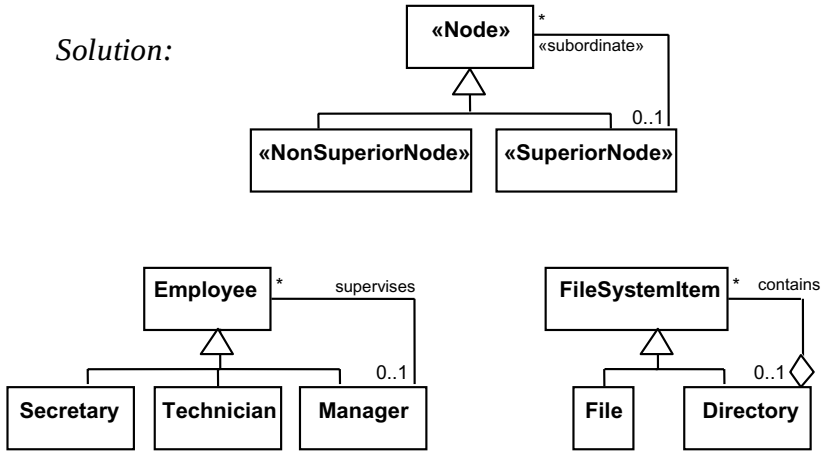
—You want a flexible way of representing the hierarchy

- that prevents certain objects from having subordinates

—All the objects have many common properties and operations

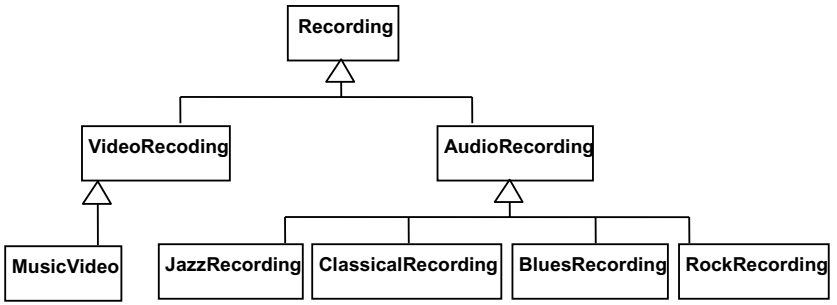
General Hierarchy

Solution:



General Hierarchy

Antipattern:



6.4 The Player-Role Pattern

Context:

—A *role* is a particular set of properties associated with an object in a particular context.

—An object may *play* different roles in different contexts.

Problem:

—How do you best model players and roles so that a player can change roles or possess multiple roles?

Player-Role

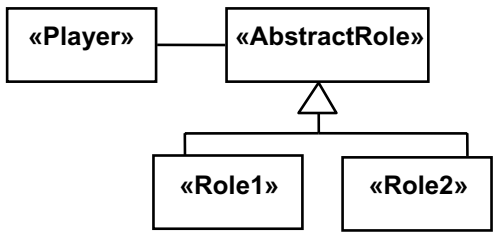
Forces:

—It is desirable to improve encapsulation by capturing the information associated with each separate role in a class.

—You want to avoid multiple inheritance.

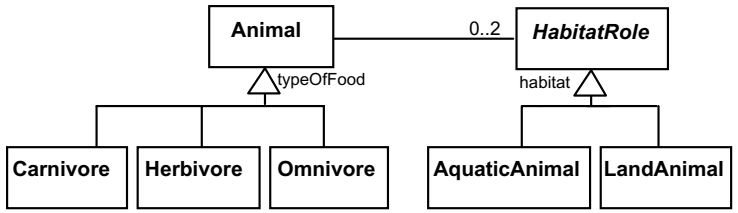
—You cannot allow an instance to change class

Solution:



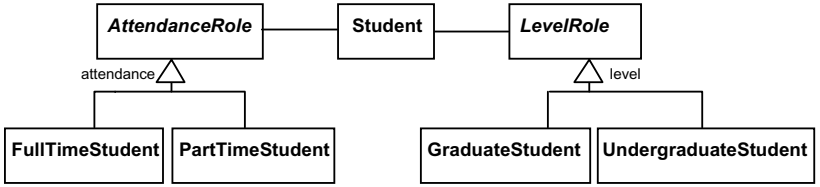
Player-Role

Example 1:



Player-Role

Example 2:



Player-Role

Antipatterns:

Merge all the properties and behaviours into a single «Player» class and not have «Role» classes at all.

Create roles as subclasses of the «Player» class.

6.5 The Singleton Pattern

Context:

—It is very common to find classes for which only one instance should exist (*singleton)*

Problem:

—How do you ensure that it is never possible to create more than one instance of a singleton class?

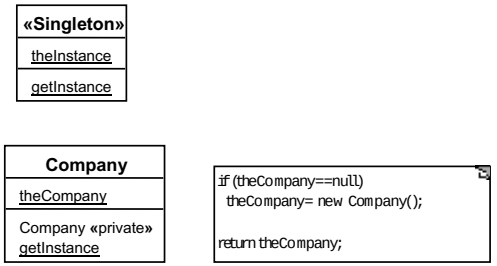
Forces:

—The use of a public constructor cannot guarantee that no more than one instance will be created.

—The singleton instance must also be accessible to all classes that require it

Singleton

Solution:



6.6 The Observer Pattern

Context:

—When an association is created between two classes, the code for the classes becomes inseparable.

—If you want to reuse one class, then you also have to reuse the other.

Problem:

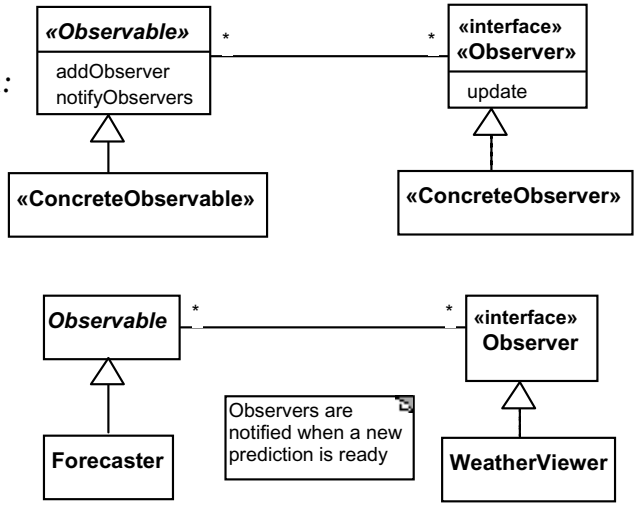
—How do you reduce the interconnection between classes, especially between classes that belong to different modules or subsystems?

Forces:

—You want to maximize the flexibility of the system to the greatest extent possible

Observer

Solution:



Observer

Antipatterns:

Connect an observer directly to an observable so that they both have references to each other.

Make the observers *subclasses* of the observable.

6.7 The Delegation Pattern

Context:

—You are designing a method in a class

—You realize that another class has a method which provides the required service

—Inheritance is not appropriate

- E.g. because the isa rule does not apply

Problem:

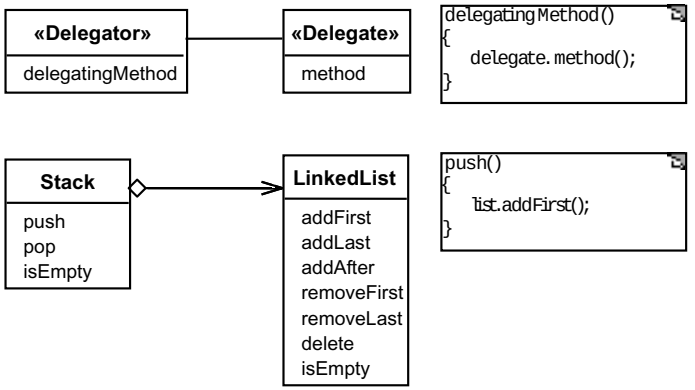
—How can you most effectively make use of a method that already exists in the other class?

Forces:

—You want to minimize development cost by reusing methods

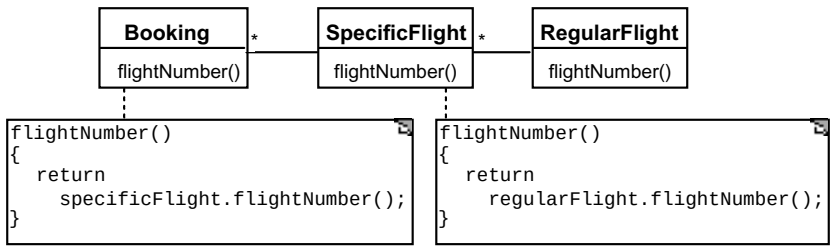
Delegation

Solution:



Delegation

Example:



Delegation

Antipatterns

Overuse generalization and *inherit* the method that is to be reused

Instead of creating a *single* method in the «Delegator» that does nothing other than call a method in the «Delegate

—consider having many different methods in the «Delegator» call the delegate's method

Access non-neighboring classes

```
return specificFlight.regularFlight.flightNumber();

return getRegularFlight().flightNumber();
```

6.8 The Adapter Pattern

Context:
—You are building an inheritance hierarchy and want to incorporate it into an existing class.
—The reused class is also often already part of its own inheritance hierarchy.

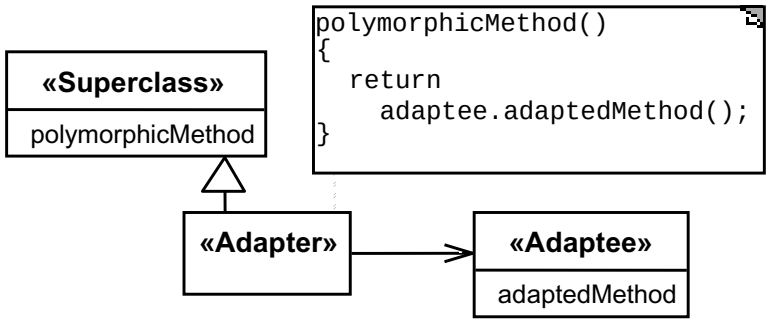
Problem:
—How to obtain the power of polymorphism when reusing a class whose methods
- have the same function
- but *not* the same signature

as the other methods in the hierarchy?

Forces:
—You do not have access to multiple inheritance or you do not want to use it.

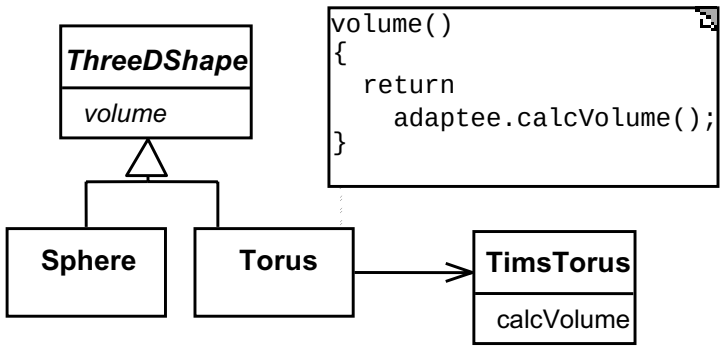
Adapter

Solution:



Adapter

Example:



6.9 The Façade Pattern

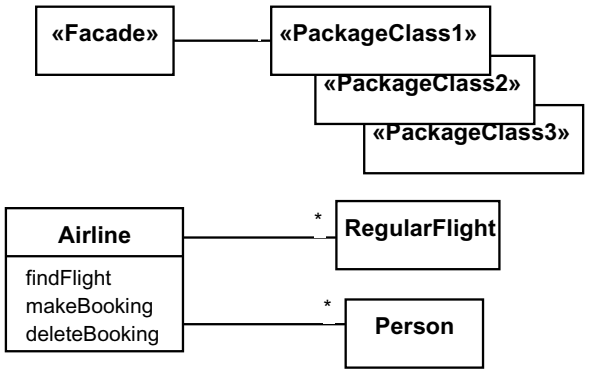
Context:
—Often, an application contains several complex packages.
—A programmer working with such packages has to manipulate many different classes

Problem:
—How do you simplify the view that programmers have of a complex package?

Forces:
—It is hard for a programmer to understand and use an entire subsystem
—If several different application classes call methods of the complex package, then any modifications made to the package will necessitate a complete review of all these classes.

Façade

Solution:



6.10 The Immutable Pattern

Context:

—An immutable object is an object that has a state that never changes after creation

Problem:

—How do you create a class whose instances are immutable?

Forces:

—There must be no loopholes that would allow 'illegal' modification of an immutable object

Solution:

—Ensure that the constructor of the immutable class is the *only* place where the values of instance variables are set or modified.

—Instance methods which access properties must not have side effects.

—If a method that would otherwise modify an instance variable is required, then it has to return a *new* instance of the class.

6.11 The Read-only Interface Pattern

Context:

—You sometimes want certain privileged classes to be able to modify attributes of objects that are otherwise immutable

Problem:

—How do you create a situation where some classes see a class as read-only whereas others are able to make modifications?

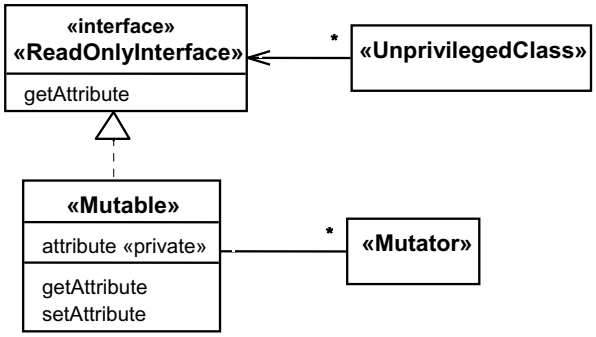
Forces:

—Restricting access by using the **public**, **protected** and **private** keywords is not adequately selective.

—Making access **public** makes it public for both reading and writing

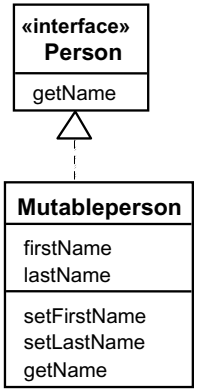
Read-only Interface

Solution:



Read-only Interface

Example:



Read-only Interface

Antipatterns:

Make the read-only class a *subclass* of the «Mutable» class

Override all methods that modify properties

—such that they throw an exception

6.12 The Proxy Pattern

Context:

—Often, it is time-consuming and complicated to create instances of a class (*heavyweight* classes).

—There is a time delay and a complex mechanism involved in creating the object in memory

Problem:

—How to reduce the need to create instances of a heavyweight class?

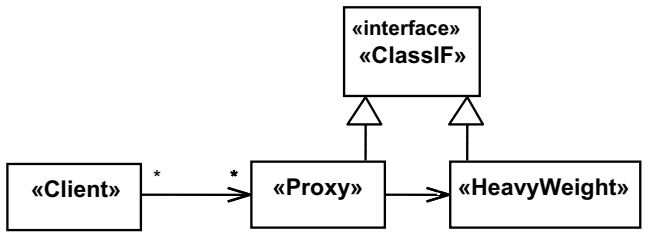
Forces:

—We want all the objects in a domain model to be available for programs to use when they execute a system's various responsibilities.

—It is also important for many objects to persist from run to run of the same program

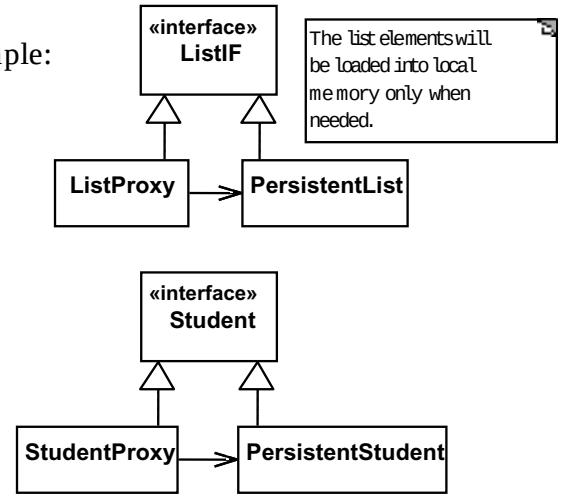
Proxy

Solution:

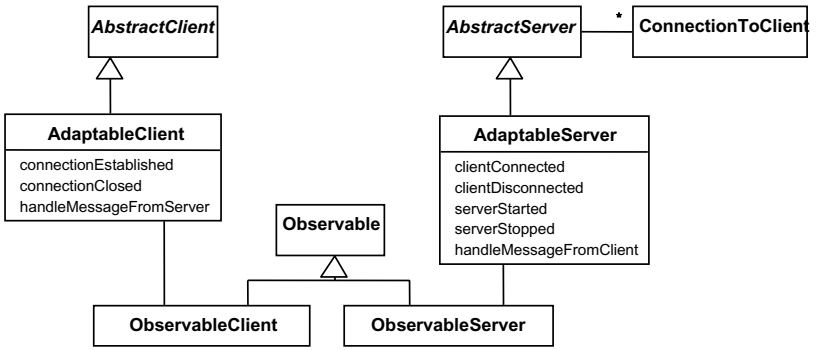


Proxy

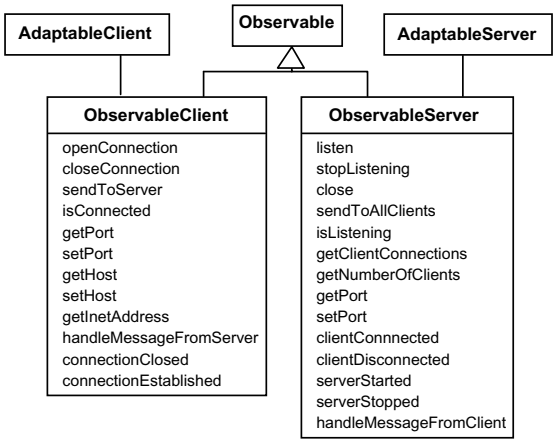
Example:



6.13 Detailed Example: The Observable layer of OCSF



The Observable layer of OCSF (continued)



Using the observable layer

1. Create a class that implements the **`Observer`** interface.
2. Register it as an observer of the **`Observable`**:

```
public MessageHandler(Observable client)
{
  client.addObserver(this);
  ...
}
```

3. Define the **`update`** method in the new class:

```
public void update(Observable obs, Object message)
{
 if (message instanceOf SomeClass)
 {
  // process the message
 }
}
```

6.14 Difficulties and Risks When Creating Class Diagrams

Patterns are not a panacea:

—Whenever you see an indication that a pattern should be applied, you might be tempted to blindly apply the pattern. However this can lead to unwise design decisions .

Resolution:

— *Always understand in depth the forces that need to be balanced, and when other patterns better balance the forces.*

—*Make sure you justify each design decision carefully.*

Difficulties and Risks When Creating Class Diagrams

Developing patterns is hard

—Writing a good pattern takes considerable work.

—A poor pattern can be hard to apply correctly

Resolution:

—*Do not write patterns for others to use until you have considerable experience both in software design and in the use of patterns.*

—*Take an in-depth course on patterns.*

—*Iteratively refine your patterns, and have them peer reviewed at each iteration.*

Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapter 7:

Focusing on Users and Their Tasks

7.1 User Centred Design

Software development should focus on the needs of users

- Understand your users
- Design software based on an understanding of the users' tasks
- Ensure users are involved in decision making processes
- Design the user interface following guidelines for good usability
- Have users work with and give their feedback about prototypes, on-line help and draft user manuals

The importance of focusing on users

- Reduced training and support costs
- Reduced time to learn the system
- Greater efficiency of use
- Reduced costs by only developing features that are needed
- Reduced costs associated with changing the system later
- Better prioritizing of work for iterative development
- Greater attractiveness of the system, so users will be more willing to buy and use it

7.2 Characteristics of Users

Software engineers must develop an understanding of the users

- Goals for using the system
- Potential patterns of use
- Demographics
- Knowledge of the domain and of computers
- Physical ability
- Psychological traits and emotional feelings

7.3 Developing Use-Case Models of Systems

A *use case* is a typical sequence of actions that a user performs in order to complete a given task

The objective of *use case analysis* is to model the system

… from the point of view of how users interact with this system

… when trying to achieve their objectives.

A *use case model* consists of

— a set of use cases

— an optional description or diagram indicating how they are related

Use cases

In general, a use case should cover the *full sequence of steps* from the beginning of a task until the end.

A use case should describe the *user's interaction* with the system ...

—not the computations the system performs.

A use case should be written so as to be as *independent* as possible from any particular user interface design.

A use case should only include actions in which the actor interacts with the computer.

Scenarios

A scenario is an *instance* of a use case

It expresses a *specific occurrence* of the use case

—a specific actor ...

—at a specific time ...

—with specific data.

How to describe a single use case

A. Name: Give a short, descriptive name to the use case.

B. Actors: List the actors who can perform this use case.

C. Goals: Explain what the actor or actors are trying to achieve.

D. Preconditions: State of the system before the use case.

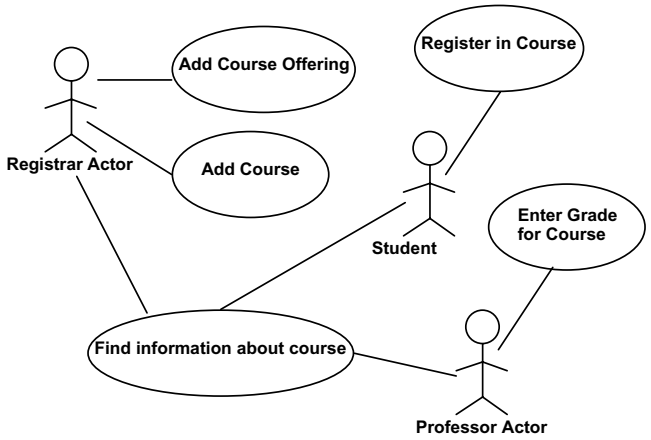
E. Description: Give a short informal description.

F. Related use cases.

G. Steps: Describe each step using a 2-column format.

H. Postconditions: State of the system in following completion.

Use case diagrams



Extensions

Used to make *optional* interactions explicit or to handle *exceptional* cases.

By creating separate use case extensions, the description of the basic use case remains simple.

A use case extension must list all the steps from the beginning of the use case to the end.

—Including the handling of the unusual situation.

Generalizations

Much like superclasses in a class diagram.

A generalized use case represents *several similar* use cases.

One or more specializations provides details of the similar use cases.

Inclusions

Allow one to express *commonality* between several different use cases.

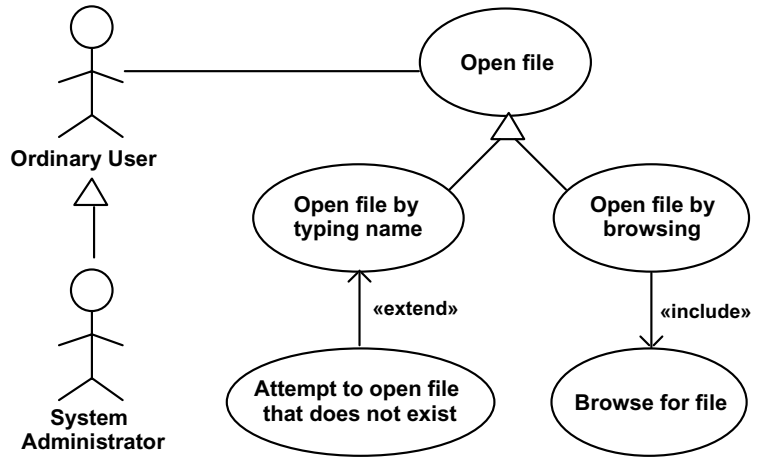
Are included in other use cases

—Even very different use cases can share sequence of actions.

—Enable you to avoid repeating details in multiple use cases.

Represent the performing of a *lower-level task* with a lower-level goal.

Example of generalization, extension and inclusion



Example description of a use case

Use case: Open file

Related use cases:
Generalization of:
Open file by typing name
Open file by browsing

Steps:

Actor actions	System responses
1. Choose 'Open…' command	2. File open dialog appears
3. Specify filename	
4. Confirm selection	5. Dialog disappears

Example (continued)

Use case: Open file by typing name

Related use cases:
Specialization of: Open file

Steps:

Actor actions	System responses
1. Choose 'Open…' command	2. File open dialog appears
3a. Select text field	
3b. Type file name	
4. Click 'Open'	5. Dialog disappears

Example (continued)

Use case: Open file by browsing

Related use cases:
Specialization of: Open file
Includes: Browse for file

Steps:

Actor actions	System responses
1. Choose 'Open…' command	2. File open dialog appears
3. Browse for file	
4. Confirm selection	5. Dialog disappears

Example (continued)

Use case: Attempt to open file that does not exist

Related use cases:
Extension of: Open file by typing name

Actor actions	System responses
1. Choose 'Open…' command	2. File open dialog appears
3a. Select text field	
3b. Type file name	
4. Click 'Open'	5. System indicates that file does not exist
6. Correct the file name	
7. Click 'Open'	8 Dialog disappears

Example (continued)

Use case: Open file by browsing (inclusion)

Steps:

Actor actions	System responses
1. If the desired file is not displayed, select a directory	2. Contents of directory is displayed
3. Repeat step 1 until the desired file is displayed	
4. Select a file	

The modeling processes: Choosing use cases on which to focus

Often one use case (or a very small number) can be identified as *central* to the system

—The entire system can be built around this particular use case

There are other reasons for focusing on particular use cases:

—Some use cases will represent a high *risk* because for some reason their implementation is problematic

—Some use cases will have high political or commercial value

The benefits of basing software development on use cases

They can help to define the *scope* of the system

They are often used to *plan* the development process

They are used to both develop and validate the requirements

They can form the basis for the definition of testcases

They can be used to structure user manuals

Use cases must not be seen as a panacea

The use cases themselves must be validated
—Using the requirements validation methods.

There are some aspects of software that are not covered by use case analysis.

Innovative solutions may not be considered.

7.4 Basics of User Interface Design

User interface design should be done in conjunction with other software engineering activities.

Do use case analysis to help define the tasks that the UI must help the user perform.

Do *iterative* UI prototyping to address the use cases.

Results of prototyping will enable you to finalize the requirements.

Usability vs. Utility

Does the system provide the *raw capabilities* to allow the user to achieve their goal?

This is *utility*.

Does the system allow the user to *learn and to use* the raw capabilities *easily*?

This is *usability*.

Both utility and usability are essential

They must be measured in the context of particular types of users.

Aspects of usability

Usability can be divided into separate aspects:

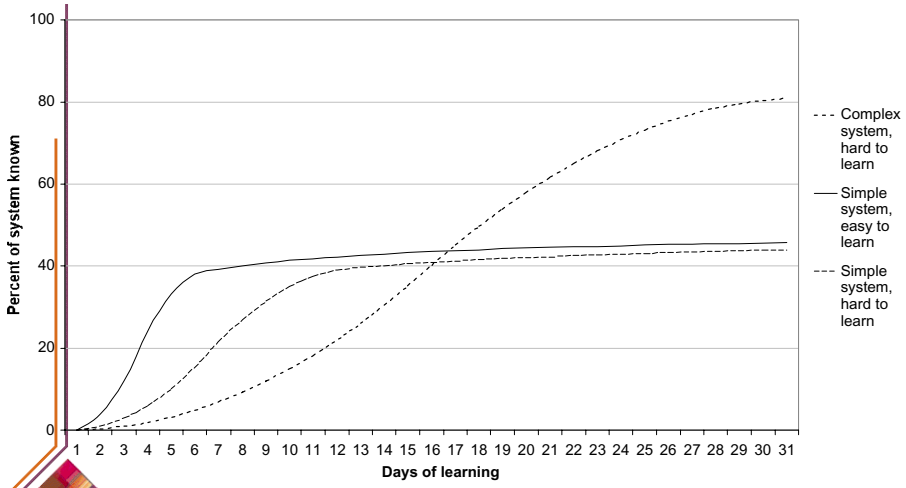
Learnability
—The speed with which a new user can become proficient with the system.

Efficiency of use
—How fast an expert user can do their work.

Error handling
—The extent to which it prevents the user from making errors, detects errors, and helps to correct errors.

Acceptability.
—The extent to which users *like* the system.

Different learning curves



Some basic terminology of user interface design

Dialog: A specific window with which a user can interact, but which is not the main UI window.

Control or Widget: Specific components of a user interface.

Affordance: The set of operations that the user can do at any given point in time.

State: At any stage in the dialog, the system is displaying certain information in certain widgets, and has a certain affordance.

Mode: A situation in which the UI restricts what the user can do.

Modal dialog: A dialog in which the system is in a very restrictive mode.

Feedback: The *response from the system* whenever the user does something, is called feedback.

Encoding techniques. Ways of encoding information so as to communicate it to the user.

6.5 Usability Principles

1. Do not rely only on usability guidelines – *always test with users.*

   Usability guidelines have exceptions; you can only be confident that a UI is good if you test it successfully with users.

2: Base UI designs on users' *tasks.*

   Perform use case analysis to structure the UI.

3: Ensure that the sequences of actions to achieve a task are as *simple* as possible.

   Reduce the amount of reading and manipulation the user has to do.

   Ensure the user does not have to navigate anywhere to do subsequent steps of a task.

Usability Principles

4: Ensure that the user always knows what he or she can and should do next.

   Ensure that the user can see *what commands are available* and are not available.

   Make the *most important commands stand out.*

5: Provide good feedback including effective error messages.

   Inform users of the *progress* of operations and of their *location* as they navigate.

   When something goes wrong explain the situation in adequate detail and *help the user to resolve the problem.*

Usability Principles

6: Ensure that the user can always get out, go back or undo an action.

- Ensure that all operations can be *undone*.
- Ensure it is easy to *navigate back* to where the user came from.

7: Ensure that response time is adequate.

- Users are very sensitive to slow response time
  - —They compare your system to others.
- Keep response time less than a second for most operations.
- Warn users of longer delays and inform them of progress.

Usability Principles

8: Use *understandable encoding techniques*.

- Choose encoding techniques with care.
- Use labels to ensure all encoding techniques are fully understood by users.

9: Ensure that the UI's appearance is *uncluttered*.

- Avoid displaying too much information.
- Organize the information effectively.

Usability Principles

10: Consider the needs of *different groups* of users.

- Accommodate people from different *locales* and people with *disabilities*.
- Ensure that the system is usable by both *beginners* and *experts*.

11: Provide all necessary *help*.

- Organize help well.
- Integrate help with the application.
- Ensure that the help is accurate.

Usability Principles

12. Be *consistent*.

- Use similar layouts and graphic designs throughout your application.
- Follow look-and-feel standards.
- Consider mimicking other applications.

Some encoding techniques

- Text and fonts
- Icons
- Photographs
- Diagrams and abstract graphics
- Colours
- Grouping and bordering
- Spoken words
- Music
- Other sounds
- Animations and video
- Flashing

Example (bad UI)

Example (better UI)

7.6 Evaluating User Interfaces

Heuristic evaluation

1. Pick some use cases to evaluate.

2. For each window, page or dialog that appears during the execution of the use case
   —Study it in detail to look for possible usability defects.

3. When you discover a usability defect write down the following information:
   —A short description of the defect.
   —Your ideas for how the defect might be fixed.

Evaluating User Interfaces

Evaluation by observation of users

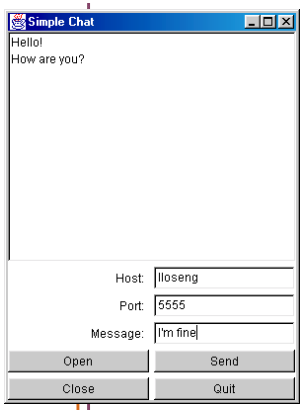
- Select users corresponding to each of the most important actors
- Select the most important use cases
- Write sufficient instructions about each of the scenarios
- Arrange evaluation sessions with users
- Explain the purpose of the evaluation
- Preferably videotape each session
- Converse with the users as they are performing the tasks
- When the users finish all the tasks, de-brief them
- Take note of any difficulties experienced by the users
- Formulate recommended changes

7.7 Implementing a Simple GUI in Java

The Abstract Window Toolkit (AWT)

- Component: the basic building blocks of any graphical interface.
  - `Button`, `TextField`, `List`, `Label`, `ScrollBar`.
- Container: contain the components constituting the GUI
  - `Frame`, `Dialog` and `Panel`
- LayoutManager: define the way components are laid out in a container.
  - `GridLayout`, `BorderLayout`

Example




```
public class ClientGUI
  extends Frame implements ChatIF
{
  private Button closeB = new Button("Close");
  private Button openB = new Button("Open");
  private Button sendB = new Button("Send");
  private Button quitB = new Button("Quit");
  private TextField portTxF = new TextField("");
  private TextField hostTxF = new TextField("");
  private TextField message = new TextField();
  private Label portLB =
    new Label("Port: ", Label.RIGHT);
  private Label hostLB =
    new Label("Host: ", Label.RIGHT);
  private Label messageLB =
    new Label("Message: ", Label.RIGHT);
  private List messageList = new List();
 ...
}
```


Example

```
public ClientGUI(String host, int port)
{
  super("Simple Chat");
  setSize(300,400);
  setVisible(true);

  setLayout(new BorderLayout(5,5));
  Panel bottom = new Panel();
  add("Center", messageList);
  add("South", bottom);
  bottom.setLayout(new GridLayout(5,2,5,5))
  bottom.add(hostLB);
  bottom.add(hostTxF);
  bottom.add(portLB);
  bottom.add(portTxF);
  bottom.add( messageLB);
  bottom.add( message);
  bottom.add(openB);
  bottom.add(sendB);
  bottom.add(closeB);
  bottom.add(quitB);
```

Example

```
sendB.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e)
  {
    send();
  }
 });
}

public void send()
{
 try
 {
  client.sendToServer(message.getText());
 }
 catch (Exception ex)
 {
   messageList.add(ex.toString());
   messageList.makeVisible(messageList.getItemCount()-1);
   messageList.setBackground(Color.yellow);
 }
}
```


7.8 Difficulties and Risks in Use Case Modelling and UI Design

Users differ widely

—*Account for differences among users when you design the system.*

—*Design it for internationalization.*

—*When you perform usability studies, try the system with many different types of users.*

User interface implementation technology changes rapidly

—*Stick to simpler UI frameworks widely used by others.*

—*Avoid fancy and unusual UI designs involving specialized controls that will be hard to change.*

Difficulties and Risks in Use Case Modelling and UI Design

User interface design and implementation can often take the majority of work in an application:

—*Make UI design an integral part of the software engineering process.*

—*Allocate time for many iterations of prototyping and evaluation.*

Developers often underestimate the weaknesses of a GUI

—*Ensure all software engineers have training in UI development.*

—*Always test with users.*

—*Study the UIs of other software.*

Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapter 8:

Modelling Interactions and Behaviour

8.1 Interaction Diagrams

Interaction diagrams are used to model the dynamic aspects of a software system

- They help you to visualize how the system runs.
- An interaction diagram is often built from a use case and a class diagram.
  - —The objective is to show how a set of objects accomplish the required interactions with an actor.

Interactions and messages

Interaction diagrams show how a set of actors and objects communicate with each other to perform:

- —The steps of a use case, or
- —The steps of some other piece of functionality.

The set of steps, taken together, is called an *interaction*.

Interaction diagrams can show several different types of communication.

- —E.g. method calls, messages send over the network
- —These are all referred to as *messages*.

Elements found in interaction diagrams

Instances of classes

- —Shown as boxes with the class and object identifier underlined

Actors

- —Use the stick-person symbol as in use case diagrams

Messages

- —Shown as arrows from actor to object, or from object to object

Creating instances diagrams

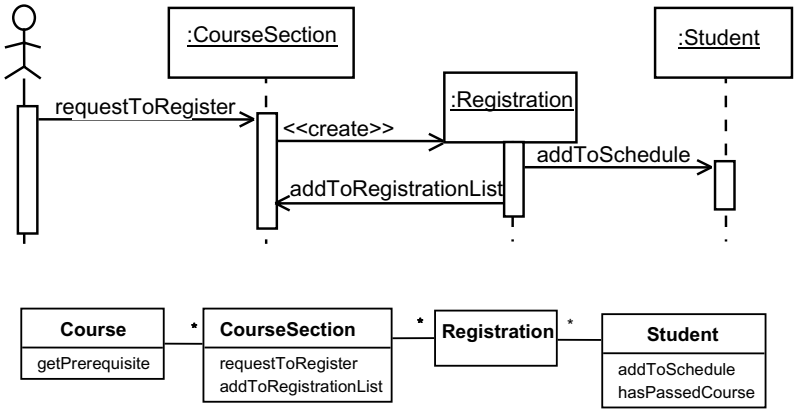
You should develop a class diagram and a use case model before starting to create an interaction diagram.

There are two kinds of interaction diagrams:

—*Sequence diagrams*

—*Collaboration diagrams*

Sequence diagrams – an example



Sequence diagrams

A sequence diagram shows the sequence of messages exchanged by the set of objects performing a certain task

The objects are arranged horizontally across the diagram.

An actor that initiates the interaction is often shown on the left.

The vertical dimension represents time.

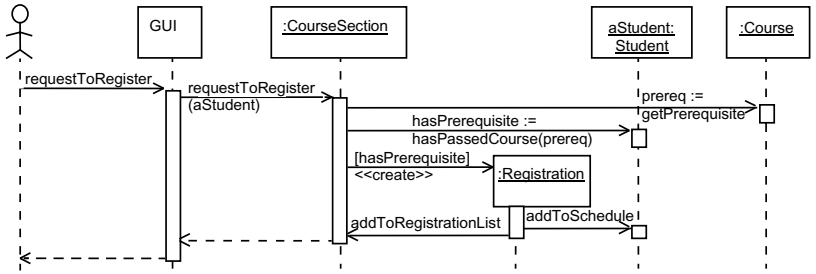
A vertical line, called a *lifeline*, is attached to each object or actor.

The lifeline becomes a broad box, called an *activation box* during the *live activation* period.

A message is represented as an arrow between activation boxes of the sender and receiver.

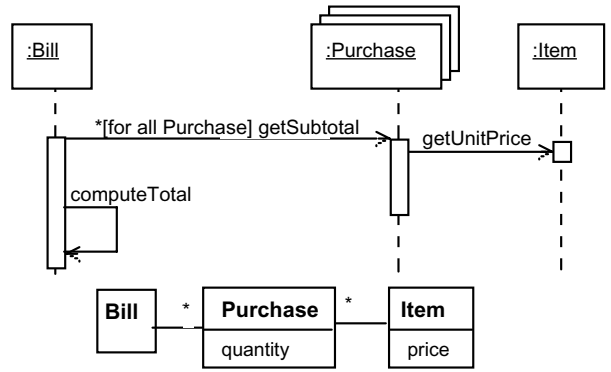
—A message is labelled and can have an argument list and a return value.

Sequence diagrams – same example, more details



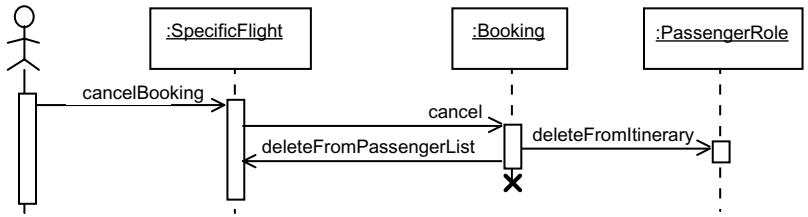
Sequence diagrams – an example with replicated messages

An *iteration* over objects is indicated by an asterisk preceding the message name

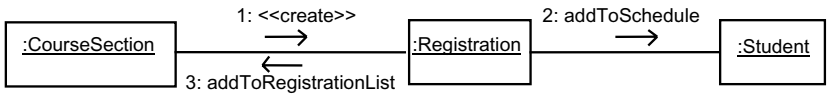


Sequence diagrams – an example with object deletion

If an object's life ends, this is shown with an X at the end of the lifeline



Collaboration diagrams – an example

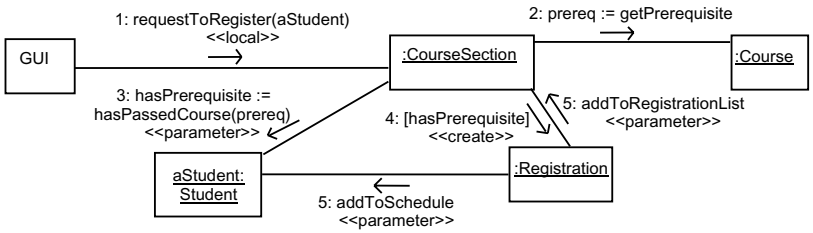


Collaboration diagrams

Collaboration diagrams emphasise how the objects collaborate in order to realize an interaction

- A collaboration diagram is a graph with the objects as the vertices.
- Communication links are added between objects
- Messages are attached to these links.
  - —Shown as arrows labelled with the message name
- Time ordering is indicated by prefixing the message with some numbering scheme.

Collaboration diagrams – same example, more details



Communication links

A communication link can exist between two objects whenever it is possible for one object to send a message to the other one.

Several situations can make this message exchange possible:

1. The classes of the two objects have an *association* between them.
   - This is the most common case.
   - If all messages are sent in the same direction, then probably the association can be made unidirectional.

Other communication links

2. The receiving object is stored in a *local* variable of the sending method.
   - This often happens when the object is created in the sending method or when some computation returns an object .
   - The stereotype to be used is «local» or [L].

3. A reference to the receiving object has been received as a *parameter* of the sending method.
   - The stereotype is «parameter» or [P].

Other communication links

4. The receiving object is global.
   - This is the case when a reference to an object can be obtained using a static method.
   - The stereotype «global», or a [G] symbol is used in this case.

5. The objects communicate over a network.
   - We suggest to write «network».

How to choose between using a sequence or collaboration diagram

Sequence diagrams

Make explicit the time ordering of the interaction.

—Use cases make time ordering explicit too

—So sequence diagrams are a natural choice when you build an interaction model from a use case.

Make it easy to add details to messages.

—Collaboration diagrams have less space for this

How to choose between using a sequence or collaboration diagram

Collaboration diagrams

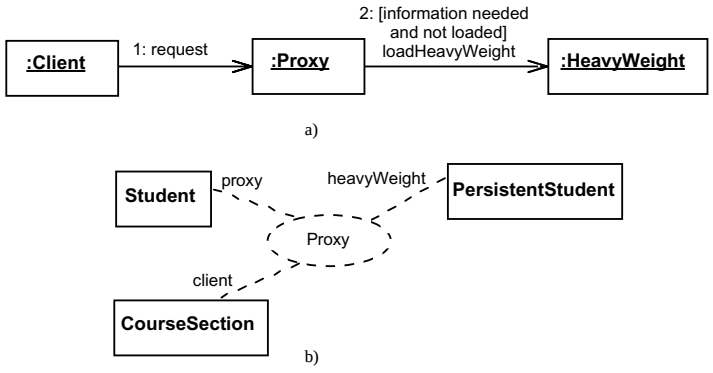
Can be seen as a projection of the class diagram

—Might be preferred when you are *deriving* an interaction diagram from a class diagram.

—Are also useful for *validating* class diagrams.

Collaboration diagrams and patterns

A collaboration diagram can be used to represent aspects of a *design pattern*



8.2 State Diagrams

A state diagram describes the behaviour of a *system*, some *part* of a system, or an *individual object*.

At any given point in time, the system or object is in a certain state.

—Being in a state means that it is will behave in a *specific way* in response to any events that occur.

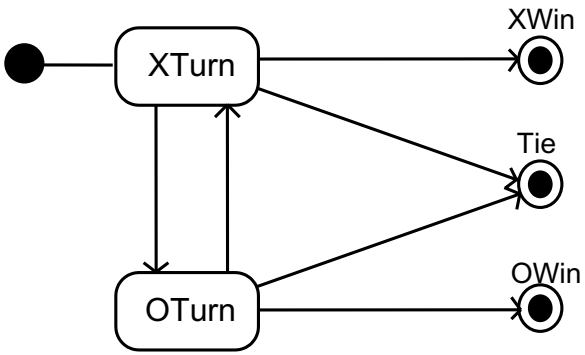
Some events will cause the system to change state.

—In the new state, the system will behave in a different way to events.

A state diagram is a directed graph where the nodes are states and the arcs are transitions.

State diagrams – an example

tic-tac-toe game



States

At any given point in time, the system is in one state.

It will remain in this state until an event occurs that causes it to change state.

A state is represented by a rounded rectangle containing the name of the state.

Special states:

—A black circle represents the *start state*

—A circle with a ring around it represents an *end state*

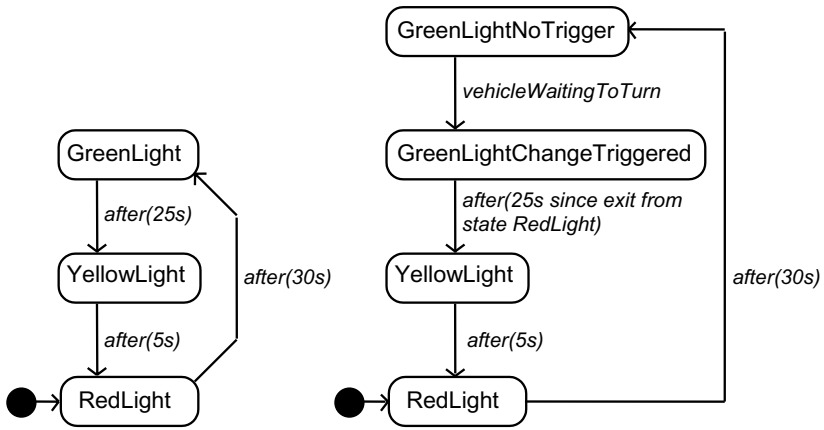
Transitions

A transition represents a change of state in response to an event.

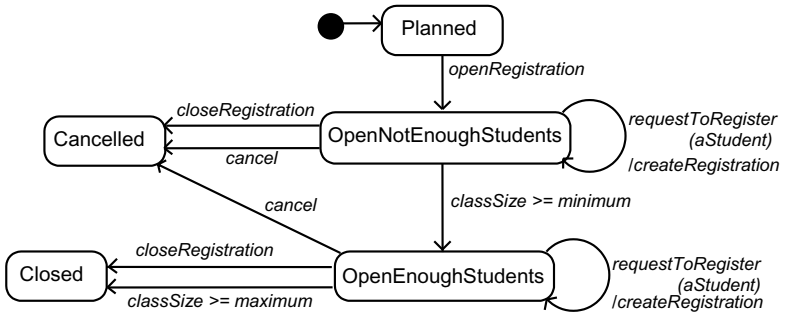
—It is considered to occur instantaneously.

The label on each transition is the event that causes the change of state.

State diagrams – an example of transitions with time-outs and conditions



State diagrams – an example with conditional transitions



Activities in state diagrams

An *activity* is something that takes place while the system is *in* a state.

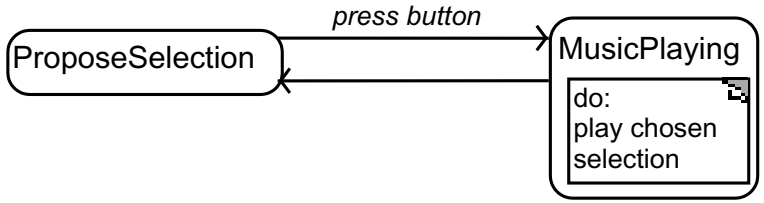
—It takes a period of time.

—The system may take a transition out of the state in response to completion of the activity,

—Some other outgoing transition may result in:

- The interruption of the activity, and
- An early exit from the state.

State diagram – an example with activity



Actions in state diagrams

An *action* is something that takes place effectively *instantaneously*

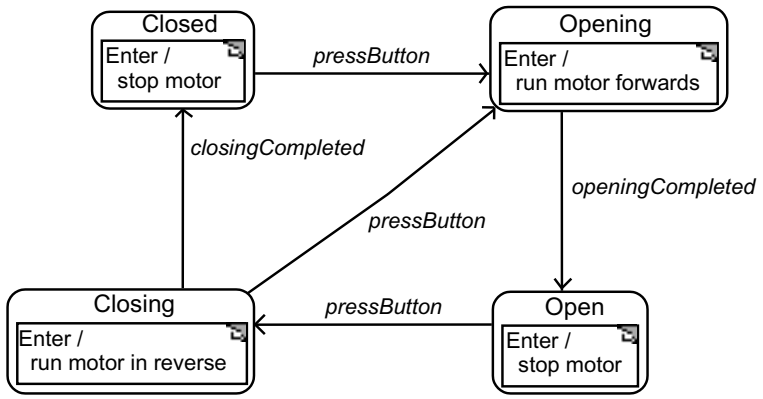
—When a particular transition is taken,

—Upon entry into a particular state, or

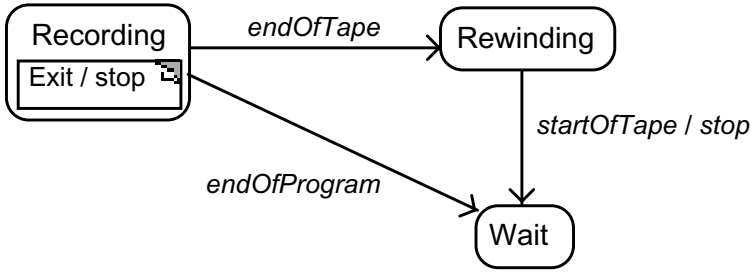
—Upon exit from a particular state

An action should consume no noticeable amount of time

State diagram – an example with actions



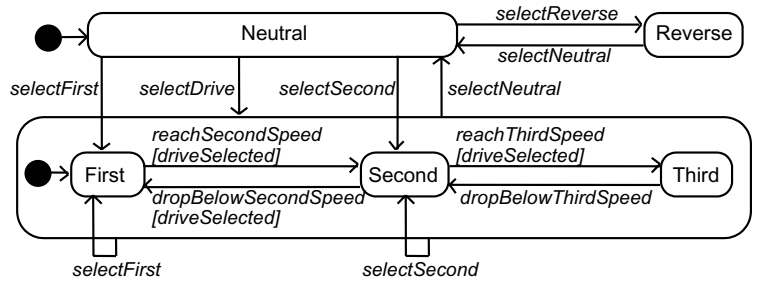
State diagrams – another example



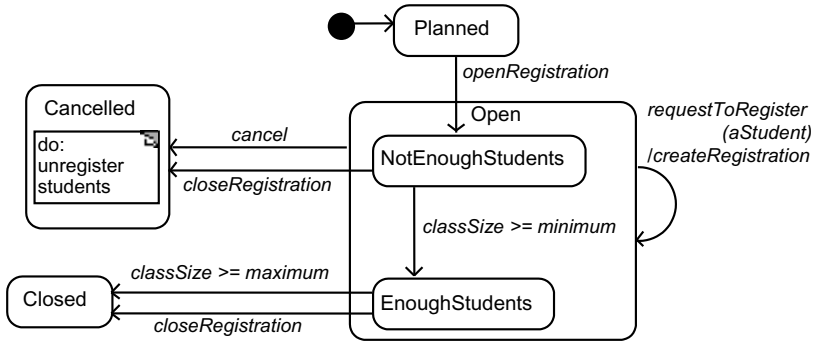
Nested substates and guard conditions

A state diagram can be nested inside a state.

The states of the inner diagram are called *substates*.



State diagram – an example with substates



8.3 Activity Diagrams

An *activity diagram* is like a state diagram.

—Except most transitions are caused by *internal* events, such as the completion of a computation.

An activity diagram

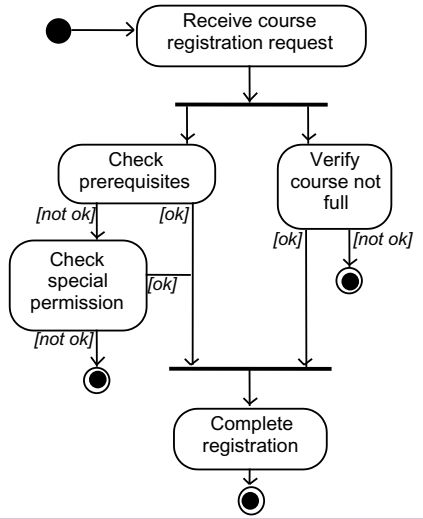
—Can be used to understand the flow of work that an object or component performs.

—Can also be used to visualize the interrelation and interaction between different use cases.

—Is most often associated with several classes.

One of the strengths of activity diagrams is the representation of *concurrent* activities.

Activity diagrams – an example



Representing concurrency

Concurrency is shown using forks, joins and rendezvous.

—A *fork* has one incoming transition and multiple outgoing transitions.

- The execution splits into two concurrent threads.

—A *rendezvous* has multiple incoming and multiple outgoing transitions.

- Once all the incoming transitions occur all the outgoing transitions may occur.

Representing concurrency

—A *join* has multiple incoming transitions and one outgoing transition.

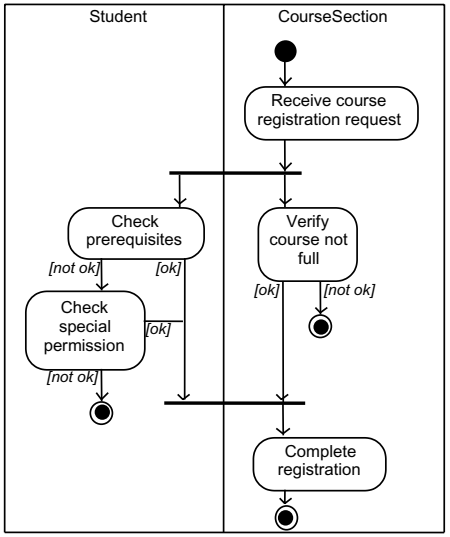
- The outgoing transition will be taken when all incoming transitions have occurred.
- The incoming transitions must be triggered in separate threads.
- If one incoming transition occurs, a wait condition occurs at the join until the other transitions occur.

Swimlanes

Activity diagrams are most often associated with several classes.

The partition of activities among the existing classes can be explicitly shown using *swimlanes*.

Activity diagrams – an example with swimlanes



8.4 Implementing Classes Based on Interaction and State Diagrams

You should use these diagrams for the parts of your system that you find most complex.

—I.e. not for every class

Interaction, activity and state diagrams help you create a correct implementation.

This is particularly true when behaviour is *distributed* across several use cases.

—E.g. a state diagram is useful when different conditions cause instances to respond differently to the same event.

Example: The **`CourseSection`** class

States:

'Planned':

- **`closedOrCancelled == false && open == false`**

'Cancelled':

- **`closedOrCancelled == true && registrationList.size() == 0`**

'Closed' (course section is too full, or being taught):

- **`closedOrCancelled == true && registrationList.size() > 0`**

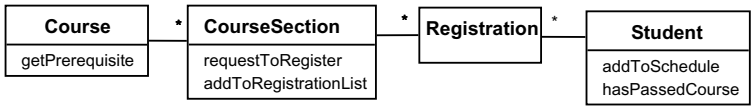
Example: The **CourseSection** class

States:

- 'Open' (accepting registrations):
  - **open == true**
- 'NotEnoughStudents' (substate of 'Open'):
  - **open == true && registrationList.size() < course.getMinimum()**
- 'EnoughStudents' (substate of 'Open'):
  - **open == true && registrationList.size() >= course.getMinimum()**

Example: The **CourseSection** class

Class diagram



Example: The **CourseSection** class

```
public class CourseSection
{
 // The many-1 abstraction-occurence association
 private Course course;

 // The 1-many association to class Registration
 private List registationList;

 // The following are present only to determine
 // the state
 // The initial state is 'Planned'
 private boolean open = false;
 private boolean closedOrCancelled = false;
 ...
}
```


Example: The **CourseSection** class

```
public CourseSection(Course course)
{
 this.course = course;
 RegistrationList = new LinkedList();
}

public void cancel()
{
 // to 'Cancelled' state
 open = false;
 closedOrCancelled = true;
 unregisterStudents();
}
```

Example: The **CourseSection** class

```
public void openRegistration()
{
 if(!closedOrCancelled)
 // must be in 'Planned' state
 {
   open = true;
  //to 'OpenNotEnoughStudents' state
 }
}
```


Example: The **CourseSection** class

```
public void closeRegistration()
{
 //to 'Cancelled' or 'Closed' state
 open = false;
 closedOrCancelled = true;
 if (registrationList.size() <
   course.getMinimum())
 {
   unregisterStudents();
  //to 'Cancelled' state
 }
}
```


Example: The **CourseSection** class

```
public void requestToRegister(Student student)
 {
  if (open)  // must be in one of the two 'Open' states
  {
   // The interaction specified in the sequence diagram
    Course prereq = course.getPrerequisite();
   if (student.hasPassedCourse(prereq))
   {
     //Indirectly calls addToRegistrationList
     new Registration(this, student);
   }

   // Check for automatic transition to 'Closed' state
   if (registrationList.size() >= course.getMaximum())
   {
    //to 'Closed' state
    open = false;
    closedOrCancelled = true;
   }
  }
 }
```


Example: The **CourseSection** class

```
// Activity associated with 'Cancelled' state.
private void unregisterStudents()
{
 Iterator it = registrationList.iterator();
 while (it.hasNext())
 {
   Registration r = (Registration)it.next();
  r.unregisterStudent();
  it.remove();
 }
}

// Called within this package only, by the
// constructor of Registration
 void addToRegistrationList(
   Registration newRegistration)
 {
  registrationList.add(newRegistration);
 }
}
```

8.5 Difficulties and Risks in Modelling Interactions and Behaviour

Dynamic modelling is a difficult skill

In a large system there are a very large number of possible paths a system can take.

It is hard to choose the classes to which to allocate each behaviour:

—*Ensure that skilled developers lead the process, and ensure that all aspects of your models are properly reviewed.*

—*Work iteratively:*

- *Develop initial class diagrams, use cases, responsibilities, interaction diagrams and state diagrams;*
- *Then go back and verify that all of these are consistent, modifying them as necessary.*

—*Drawing different diagrams that capture related, but distinct, information will often highlight problems.*

Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapter 9:

Architecting and Designing Software

9.1 The Process of Design

Definition:

Design is a problem-solving process whose objective is to find and describe a way:

—To implement the system's *functional requirements*...

—While respecting the constraints imposed by the *non-functional requirements*...

- including the budget

—And while adhering to general principles of *good quality*

Design as a series of decisions

A designer is faced with a series of *design issues*

These are sub-problems of the overall design problem.

Each issue normally has several alternative solutions:

—design *options*.

The designer makes a *design decision* to resolve each issue.

—This process involves choosing the best option from among the alternatives.

Making decisions

To make each design decision, the software engineer uses:

Knowledge of

—the requirements

—the design as created so far

—the technology available

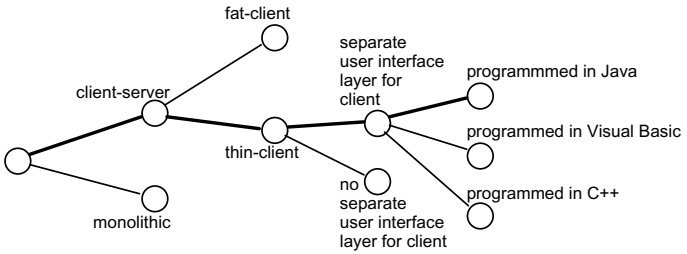
—software design principles and 'best practices'

—what has worked well in the past

Design space

The space of possible designs that could be achieved by choosing different sets of alternatives is often called the *design space*

For example:



Component

Any piece of software or hardware that has a clear role.

A component can be isolated, allowing you to replace it with a different component that has equivalent functionality.

Many components are designed to be reusable.

Conversely, others perform special-purpose functions.

Module

A component that is defined at the programming language level

For example, methods, classes and packages are modules in Java.

System

A logical entity, having a set of definable responsibilities or objectives, and consisting of hardware, software or both.

A system can have a specification which is then implemented by a collection of components.

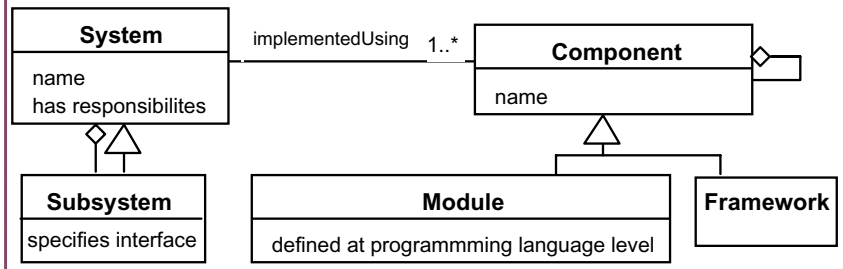
A system continues to exist, even if its components are changed or replaced.

The goal of requirements analysis is to determine the responsibilities of a system.

Subsystem:

—A system that is part of a larger system, and which has a definite interface

UML diagram of system parts



Top-down and bottom-up design

Top-down design

- First design the very high level structure of the system.
- Then gradually work down to detailed decisions about low-level constructs.
- Finally arrive at detailed decisions such as:
  - —the format of particular data items;
  - —the individual algorithms that will be used.

Top-down and bottom-up design

Bottom-up design

- Make decisions about reusable low-level utilities.
- Then decide how these will be put together to create high-level constructs.

A mix of top-down and bottom-up approaches are normally used:

- Top-down design is almost always needed to give the system a good structure.
- Bottom-up design is normally useful so that reusable components can be created.

Different aspects of design

Architecture design:
- —The division into subsystems and components,
  - How these will be connected.
  - How they will interact.
  - Their interfaces.

Class design:
- —The various features of classes.

User interface design

Algorithm design:
- —The design of computational mechanisms.

Protocol design:
- —The design of communications protocol.

9.2 Principles Leading to Good Design

Overall *goals* of good design:

- Increasing profit by reducing cost and increasing revenue
- Ensuring that we actually conform with the requirements
- Accelerating development
- Increasing qualities such as
  - —Usability
  - —Efficiency
  - —Reliability
  - —Maintainability
  - —Reusability

Design Principle 1: Divide and conquer

Trying to deal with something big all at once is normally much harder than dealing with a series of smaller things

- Separate people can work on each part.
- An individual software engineer can specialize.
- Each individual component is smaller, and therefore easier to understand.
- Parts can be replaced or changed without having to replace or extensively change other parts.

Ways of dividing a software system

A distributed system is divided up into clients and servers

A system is divided up into subsystems

A subsystem can be divided up into one or more packages

A package is divided up into classes

A class is divided up into methods

Design Principle 2: Increase cohesion where possible

A subsystem or module has high cohesion if it keeps together things that are related to each other, and keeps out other things

- This makes the system as a whole easier to understand and change
- Type of cohesion:
  - —Functional, Layer, Communicational, Sequential, Procedural, Temporal, Utility

Functional cohesion

This is achieved when all the code that computes a particular result is kept together - and everything else is kept out

i.e. when a module only performs a *single* computation, and returns a result, *without having side-effects*.

Benefits to the system:

—Easier to understand

—More reusable

—Easier to replace

Modules that update a database, create a new file or interact with the user are not functionally cohesive

Layer cohesion

All the facilities for providing or accessing a set of related services are kept together, and everything else is kept out

The layers should form a hierarchy

—Higher layers can access services of lower layers,

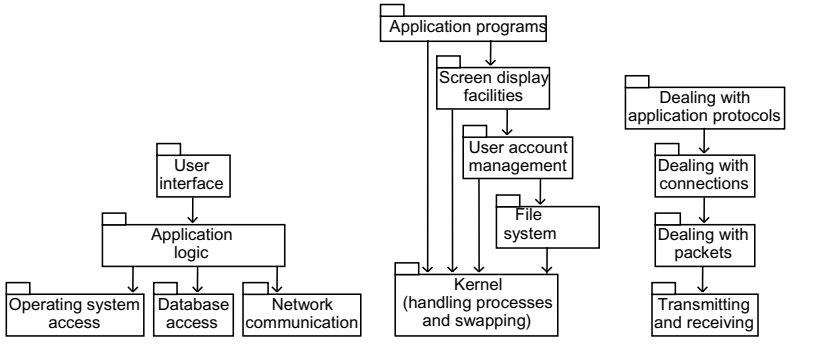
—Lower layers do not access higher layers

The set of procedures through which a layer provides its services is the *application programming interface (API)*

You can replace a layer without having any impact on the other layers

—You just replicate the API

Example of the use of layers



a) Typical layers in an application program

b) Typical layers in an operating system

c) Simplified view of layers in a communication system

Communicational cohesion

All the modules that access or manipulate certain data are kept together (e.g. in the same class) - and everything else is kept out

A class would have good communicational cohesion

—if all the system's facilities for storing and manipulating its data are contained in this class.

—if the class does not do anything other than manage its data.

Main advantage: When you need to make changes to the data, you find all the code in one place

Sequential cohesion

Procedures, in which one procedure provides input to the next, are kept together – and everything else is kept out

- You should achieve sequential cohesion, only once you have already achieved the preceding types of cohesion.

Procedural cohesion

Keep together several procedures that are used one after another

- Even if one does not necessarily provide input to the next.
- Weaker than sequential cohesion.

Temporal Cohesion

Operations that are performed during the same phase of the execution of the program are kept together, and everything else is kept out

- For example, placing together the code used during system start-up or initialization.
- Weaker than procedural cohesion.

Utility cohesion

When related utilities which cannot be logically placed in other cohesive units are kept together

- A utility is a procedure or class that has wide applicability to many different subsystems and is designed to be reusable.
- For example, the **`java.lang.Math`** class.

Design Principle 3: Reduce coupling where possible

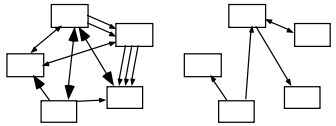
Coupling occurs when there are *interdependencies* between one module and another

When interdependencies exist, changes in one place will require changes somewhere else.

A network of interdependencies makes it hard to see at a glance how some component works.

Type of coupling:

—Content, Common, Control, Stamp, Data, Routine Call, Type use, Inclusion/Import, External



Content coupling:

Occurs when one component *surreptitiously* modifies data that is *internal* to another component

To reduce content coupling you should therefore *encapsulate* all instance variables

—declare them `private`

—and provide get and set methods

A worse form of content coupling occurs when you directly modify an instance variable *of* an instance variable

Example of content coupling

```
public class Line
{
  private Point start, end;
  ...
  public Point getStart() { return start; }
  public Point getEnd()  { return end; }
}

public class Arch
{
  private Line baseline;
  ...
  void slant(int newY)
  {
    Point theEnd = baseline.getEnd();
    theEnd.setLocation(theEnd.getX(),newY);
  }
}
```


Common coupling

Occurs whenever you use a global variable

All the components using the global variable become coupled to each other

A weaker form of common coupling is when a variable can be accessed by a *subset* of the system's classes

—e.g. a Java package

Can be acceptable for creating global variables that represent system-wide default values

The Singleton pattern provides encapsulated global access to an object

Control coupling

Occurs when one procedure calls another using a 'flag' or 'command' that explicitly controls what the second procedure does

- To make a change you have to change both the calling and called method
- The use of polymorphic operations is normally the best way to avoid control coupling
- One way to reduce the control coupling could be to have a *look-up table*
  - —commands are then mapped to a method that should be called when that command is issued

Example of control coupling

```
public routineX(String command)
{
  if (command.equals("drawCircle")
  {
    drawCircle();
  }
  else
  {
    drawRectangle();
  }
}
```


Stamp coupling:

Occurs whenever one of your application classes is declared as the *type* of a method argument

- Since one class now uses the other, changing the system becomes harder
  - —Reusing one class requires reusing the other

- Two ways to reduce stamp coupling,
  - —using an interface as the argument type
  - —passing simple variables

Example of stamp coupling

```
public class Emailer
{
  public void sendEmail(Employee e, String text)
  {...}
  ...
}
```

Using simple data types to avoid it:

```
public class Emailer
{
  public void sendEmail(
    String name, String email, String text)
  {...}
  ...
}
```

Example of stamp coupling

Using an interface to avoid it:

```
public interface Addressee
{
  public abstract String getName();
  public abstract String getEmail();
}

public class Employee implements Addressee {…}

public class Emailer
{
  public void sendEmail(
    Addressee e, String text)
  {…}
  …
}
```


Data coupling

Occurs whenever the types of method arguments are either primitive or else simple library classes

- The more arguments a method has, the higher the coupling
  - —All methods that use the method must pass all the arguments
- You should reduce coupling by not giving methods unnecessary arguments

- There is a trade-off between data coupling and stamp coupling
  - —Increasing one often decreases the other

Routine call coupling

Occurs when one routine (or method in an object oriented system) calls another

- The routines are coupled because they depend on each other's behaviour
- Routine call coupling is always present in any system.

- If you repetitively use a sequence of two or more methods to compute something
  - —then you can reduce routine call coupling by writing a single routine that encapsulates the sequence.

Type use coupling

Occurs when a module uses a data type defined in another module

- It occurs any time a class declares an instance variable or a local variable as having another class for its type.
- The consequence of type use coupling is that if the type definition changes, then the users of the type may have to change
- Always declare the type of a variable to be the most general possible class or interface that contains the required operations

Inclusion or import coupling

Occurs when one component imports a package
(as in Java)
or when one component includes another
(as in C++).

The including or importing component is now exposed to everything in the included or imported component.

If the included/imported component changes something or adds something.

—This may raises a conflict with something in the includer, forcing the includer to change.

An item in an imported component might have the same name as something you have already defined.

External coupling

When a module has a dependency on such things as the operating system, shared libraries or the hardware

It is best to reduce the number of places in the code where such dependencies exist.

The Façade design pattern can reduce external coupling

Design Principle 4: Keep the level of abstraction as high as possible

Ensure that your designs allow you to hide or defer consideration of details, thus reducing complexity

A good abstraction is said to provide *information hiding*

Abstractions allow you to understand the essence of a subsystem without having to know unnecessary details

Abstraction and classes

Classes are data abstractions that contain procedural abstractions

Abstraction is increased by defining all variables as private.

The fewer public methods in a class, the better the abstraction

Superclasses and interfaces increase the level of abstraction

Attributes and associations are also data abstractions.

Methods are procedural abstractions

—Better abstractions are achieved by giving methods fewer parameters

Design Principle 5: Increase reusability where possible

Design the various aspects of your system so that they can be used again in other contexts

- Generalize your design as much as possible
- Follow the preceding three design principles
- Design your system to contain hooks
- Simplify your design as much as possible

Design Principle 6: Reuse existing designs and code where possible

Design with reuse is complementary to design for reusability

- Actively reusing designs or code allows you to take advantage of the investment you or others have made in reusable components
  - —*Cloning* should not be seen as a form of reuse

Design Principle 7: Design for flexibility

Actively anticipate changes that a design may have to undergo in the future, and prepare for them

- Reduce coupling and increase cohesion
- Create abstractions
- Do not hard-code anything
- Leave all options open
  - —Do not restrict the options of people who have to modify the system later
- Use reusable code and make code reusable

Design Principle 8: Anticipate obsolescence

Plan for changes in the technology or environment so the software will continue to run or can be easily changed

- Avoid using early releases of technology
- Avoid using software libraries that are specific to particular environments
- Avoid using undocumented features or little-used features of software libraries
- Avoid using software or special hardware from companies that are less likely to provide long-term support
- Use standard languages and technologies that are supported by multiple vendors

Design Principle 9: Design for Portability

Have the software run on as many platforms as possible

- Avoid the use of facilities that are specific to one particular environment
- E.g. a library only available in Microsoft Windows

Design Principle 10: Design for Testability

Take steps to make testing easier

- Design a program to automatically test the software
  - —Discussed more in Chapter 10
  - —Ensure that all the functionality of the code can by driven by an external program, bypassing a graphical user interface
- In Java, you can create a main() method in each class in order to exercise the other methods

Design Principle 10: Design defensively

Never trust how others will try to use a component you are designing

- Handle all cases where other code might attempt to use your component inappropriately
- Check that all of the inputs to your component are valid: the *preconditions*
  - —Unfortunately, over-zealous defensive design can result in unnecessarily repetitive checking

Design by contract

A technique that allows you to design defensively in an efficient and systematic way

- Key idea
  - —each method has an explicit *contract* with its callers
- The contract has a set of assertions that state:
  - —What *preconditions* the called method requires to be true when it starts executing
  - —What *postconditions* the called method agrees to ensure are true when it finishes executing
  - —What *invariants* the called method agrees will not change as it executes

9.3 Techniques for making good design decisions

Using priorities and objectives to decide among alternatives

Step 1: List and describe the alternatives for the design decision.

Step 2: List the advantages and disadvantages of each alternative with respect to your objectives and priorities.

Step 3: Determine whether any of the alternatives prevents you from meeting one or more of the objectives.

Step 4: Choose the alternative that helps you to best meet your objectives.

Step 5: Adjust priorities for subsequent decision making.

Example priorities and objectives

Imagine a system has the following objectives, starting with top priority:

Security: Encryption must not be breakable within 100 hours of computing time on a 400Mhz Intel processor, using known cryptanalysis techniques.

Maintainability. No specific objective.

CPU efficiency. Must respond to the user within one second when running on a 400MHz Intel processor.

Network bandwidth efficiency: Must not require transmission of more than 8KB of data per transaction.

Memory efficiency. Must not consume over 20MB of RAM.

Portability. Must be able to run on Windows 98, NT 4 and ME as well as Linux

Example evaluation of alternatives

	Security	Maintain-ability	Memory efficiency	CPU efficiency	Bandwidth efficiency	Portability
Algorithm A	High	Medium	High	Medium; NO	Low	Low
Algorithm B	High	High	Low	Medium; NO	Medium	Low
Algorithm C	High	High	High	Low; NO	High	Low
Algorithm D				Medium; NO	NO	
Algorithm E	NO			Low; NO		

'NO' means that the objective is not met

Using cost-benefit analysis to choose among alternatives

To estimate the *costs*, add up:

—The incremental cost of doing the *software engineering* work, including ongoing maintenance

—The incremental costs of any *development technology* required

—The incremental costs that *end-users and product support personnel* will experience

To estimate the *benefits*, add up:

—The incremental software engineering time saved

—The incremental benefits measured in terms of either increased sales or else financial benefit to users

9.4 Software Architecture

Software architecture is process of designing the global organization of a software system, including:

- Dividing software into subsystems.
- Deciding how these will interact.
- Determining their interfaces.
  - —The architecture is the core of the design, so all software engineers need to understand it.
  - —The architecture will often constrain the overall efficiency, reusability and maintainability of the system.

The importance of software architecture

Why you need to develop an architectural model:

- To enable everyone to better understand the system
- To allow people to work on individual pieces of the system in isolation
- To prepare for extension of the system
- To facilitate reuse and reusability

Contents of a good architectural model

A system's architecture will often be expressed in terms of several different *views*

- The logical breakdown into subsystems
- The interfaces among the subsystems
- The dynamics of the interaction among components at run time
- The data that will be shared among the subsystems
- The components that will exist at run time, and the machines or devices on which they will be located

Design stable architecture

To ensure the maintainability and reliability of a system, an architectural model must be designed to be *stable*.

- Being stable means that the new features can be easily added with only small changes to the architecture

Developing an architectural model

Start by sketching an outline of the architecture

- Based on the principal requirements and use cases
- Determine the main components that will be needed
- Choose among the various architectural patterns
  - —Discussed next
- *Suggestion*: have several different teams independently develop a first draft of the architecture and merge together the best ideas

Developing an architectural model

Refine the architecture

- —Identify the main ways in which the components will interact and the interfaces between them
- —Decide how each piece of data and functionality will be distributed among the various components
- —Determine if you can re-use an existing framework, if you can build a framework

Consider each use case and adjust the architecture to make it realizable

Mature the architecture

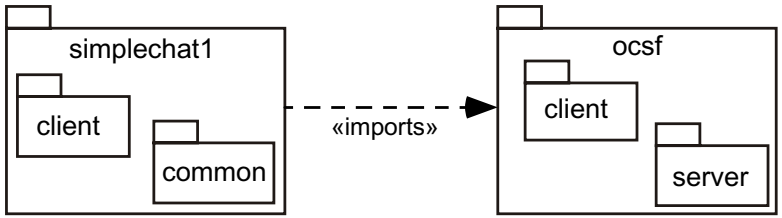
Describing an architecture using UML

All UML diagrams can be useful to describe aspects of the architectural model

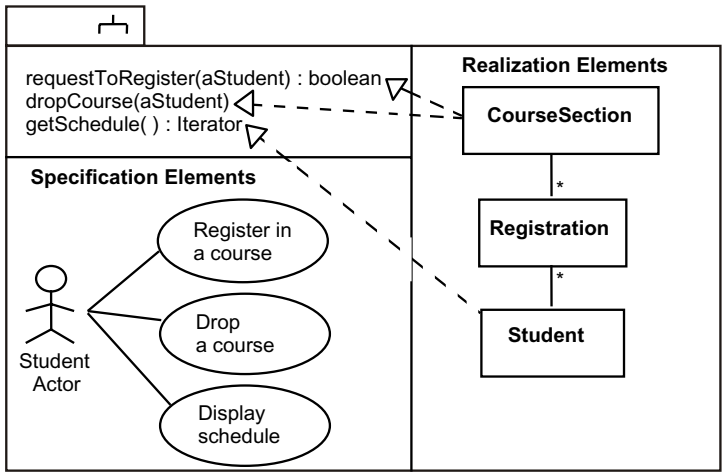
Four UML diagrams are particularly suitable for architecture modelling:

- —Package diagrams
- —Subsystem diagrams
- —Component diagrams
- —Deployment diagrams

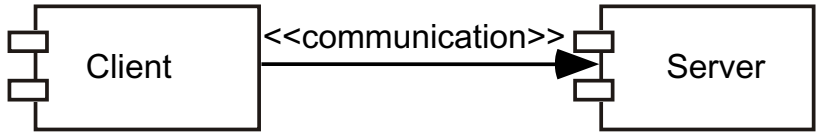
Package diagrams



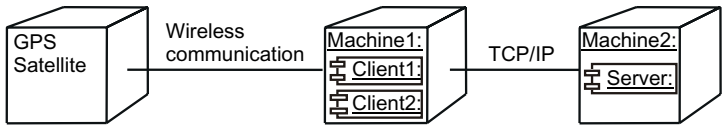
Subsystem diagrams



Component diagrams



Deployment diagrams



9.5 Architectural Patterns

The notion of patterns can be applied to software architecture.

- These are called *architectural patterns* or *architectural styles*.
- Each allows you to design flexible systems using components
  - —The components are as independent of each other as possible.

The Multi-Layer architectural pattern

In a layered system, each layer communicates only with the layer immediately below it.

Each layer has a well-defined interface used by the layer immediately above.

—The higher layer sees the lower layer as a set of *services*.

A complex system can be built by superposing layers at increasing levels of abstraction.

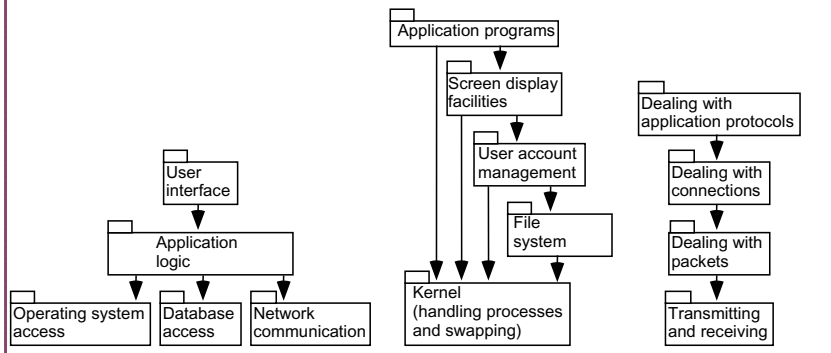
—It is important to have a separate layer for the UI.

—Layers immediately below the UI layer provide the application functions determined by the use-cases.

—Bottom layers provide general services.

- e.g. network communication, database access

Example of multi-layer systems



a) Typical layers in an application program

b) Typical layers in an operating system

c) Simplified view of layers in a communication system

The multi-layer architecture and design principles

1. *Divide and conquer*: The layers can be independently designed.
2. *Increase cohesion*: Well-designed layers have layer cohesion.
3. *Reduce coupling*: Well-designed lower layers do not know about the higher layers and the only connection between layers is through the API.
4. *Increase abstraction*: you do not need to know the details of how the lower layers are implemented.
5. *Increase reusability*: The lower layers can often be designed generically.

The multi-layer architecture and design principles

6. *Increase reuse*: You can often reuse layers built by others that provide the services you need.
7. Increase flexibility: you can add new facilities built on lower-level services, or replace higher-level layers.
8. *Anticipate obsolescence*: By isolating components in separate layers, the system becomes more resistant to obsolescence.
9. *Design for portability*: All the dependent facilities can be isolated in one of the lower layers.
10. *Design for testability*: Layers can be tested independently.
11. *Design defensively*: The APIs of layers are natural places to build in rigorous assertion-checking.

The Client-Server and other distributed architectural patterns

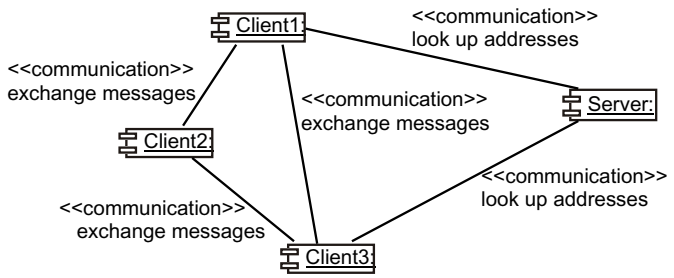
There is at least one component that has the role of *server*, waiting for and then handling connections.

There is at least one component that has the role of *client*, initiating connections in order to obtain some service.

A further extension is the Peer-to-Peer pattern.

—A system composed of various software components that are distributed over several hosts.

An example of a distributed system



The distributed architecture and design principles

1. *Divide and conquer*: Dividing the system into client and server processes is a strong way to divide the system.
   —Each can be separately developed.
2. *Increase cohesion*: The server can provide a cohesive service to clients.
3. *Reduce coupling*: There is usually only one communication channel exchanging simple messages.
4. *Increase abstraction*: Separate distributed components are often good abstractions.
6. *Increase reuse*: It is often possible to find suitable frameworks on which to build good distributed systems
   —However, client-server systems are often very application specific.

The distributed architecture and design principles

7. *Design for flexibility*: Distributed systems can often be easily reconfigured by adding extra servers or clients.
9. *Design for portability*: You can write clients for new platforms without having to port the server.
10 *Design for testability*: You can test clients and servers independently.
11. *Design defensively*: You can put rigorous checks in the message handling code.

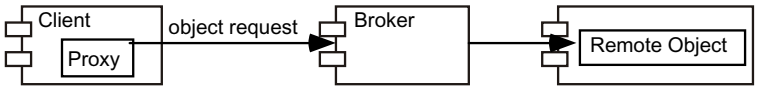
The Broker architectural pattern

Transparently distribute aspects of the software system to different nodes

—An object can call methods of another object without knowing that this object is remotely located.

—CORBA is a well-known open standard that allows you to build this kind of architecture.

Example of a Broker system



The broker architecture and design principles

1. *Divide and conquer*: The remote objects can be independently designed.
5. *Increase reusability*: It is often possible to design the remote objects so that other systems can use them too.
6. *Increase reuse*: You may be able to reuse remote objects that others have created.
7. *Design for flexibility*: The brokers can be updated as required, or the proxy can communicate with a different remote object.
9. *Design for portability*: You can write clients for new platforms while still accessing brokers and remote objects on other platforms.
11. *Design defensively*: You can provide careful assertion checking in the remote objects.

The Transaction-Processing architectural pattern

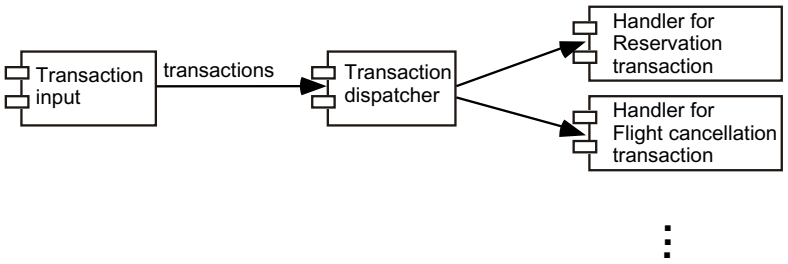
A process reads a series of inputs one by one.

Each input describes a *transaction* – a command that typically some change to the data stored by the system

There is a transaction handler component that decides what to do with each transaction

This dispatches a procedure call or message to a component that will handle the transaction

Example of a transaction-processing system



The transaction-processing architecture and design principles

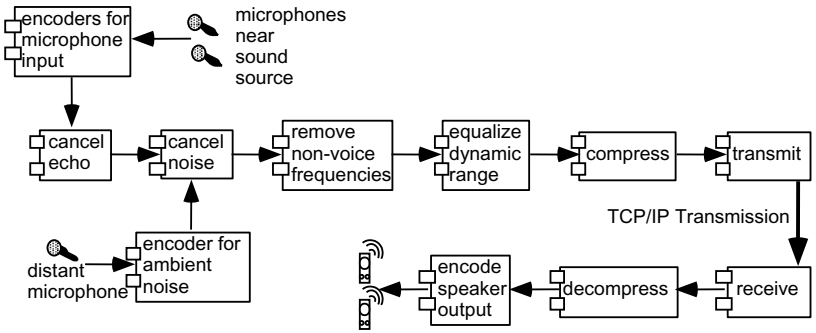
1. *Divide and conquer*: The transaction handlers are suitable system divisions that you can give to separate software engineers.
2. *Increase cohesion*: Transaction handlers are naturally cohesive units.
3. *Reduce coupling*: Separating the dispatcher from the handlers tends to reduce coupling.
7. *Design for flexibility*: You can readily add new transaction handlers.
11. *Design defensively*: You can add assertion checking in each transaction handler and/or in the dispatcher.

The Pipe-and-Filter architectural pattern

A stream of data, in a relatively simple format, is passed through a series of processes

- Each of which transforms it in some way.
- Data is constantly fed into the pipeline.
- The processes work concurrently.
- The architecture is very flexible.
  - —Almost all the components could be removed.
  - —Components could be replaced.
  - —New components could be inserted.
  - —Certain components could be reordered.

Example of a pipe-and-filter system



The pipe-and-filter architecture and design principles

1. *Divide and conquer*: The separate processes can be independently designed.
2. *Increase cohesion*: The processes have functional cohesion.
3. *Reduce coupling*: The processes have only one input and one output.
4. *Increase abstraction*: The pipeline components are often good abstractions, hiding their internal details.
5. *Increase reusability*: The processes can often be used in many different contexts.
6. *Increase reuse*: It is often possible to find reusable components to insert into a pipeline.

The pipe-and-filter architecture and design principles

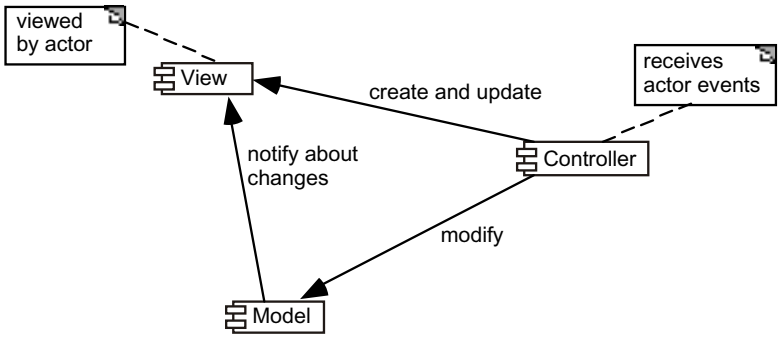
7. *Design for flexibility*: There are several ways in which the system is flexible.
10. *Design for testability*: It is normally easy to test the individual processes.
11. *Design defensively*: You rigorously check the inputs of each component, or else you can use design by contract.

The Model-View-Controller (MVC) architectural pattern

An architectural pattern used to help separate the user interface layer from other parts of the system

- The *model* contains the underlying classes whose instances are to be viewed and manipulated
- The *view* contains objects used to render the appearance of the data from the model in the user interface
- The *controller* contains the objects that control and handle the user's interaction with the view and the model
- The Observable design pattern is normally used to separate the model from the view

Example of the MVC architecture for the UI



The MVC architecture and design principles

1. *Divide and conquer*: The three components can be somewhat independently designed.
2. *Increase cohesion*: The components have stronger layer cohesion than if the view and controller were together in a single UI layer.
3. *Reduce coupling*: The communication channels between the three components are minimal.
6. *Increase reuse*: The view and controller normally make extensive use of reusable components for various kinds of UI controls.
7. *Design for flexibility*: It is usually quite easy to change the UI by changing the view, the controller, or both.
10. *Design for testability*: You can test the application separately from the UI.

9.6 Writing a Good Design Document

Design documents as an aid to making better designs

They force you to be explicit and consider the important issues before starting implementation.

They allow a group of people to review the design and therefore to improve it.

Design documents as a means of communication.

—To those who will be *implementing* the design.

—To those who will need, in the future, to *modify* the design.

—To those who need to create systems or subsystems that *interface* with the system being designed.

Structure of a design document

A. Purpose:

—What system or part of the system this design document describes.

—Make reference to the requirements that are being implemented by this design (*traceability)* .

B. General priorities:

—Describe the priorities used to guide the design process.

C. Outline of the design:

—Give a high-level description of the design that allows the reader to quickly get a general feeling for it.

D. Major design issues:

—Discuss the important issues that had to be resolved.

—Give the possible alternatives that were considered, the final decision and the rationale for the decision.

E. Other details of the design:

—Give any other details the reader may want to know that have not yet been mentioned.

When writing the document

Avoid documenting information that would be *readily obvious* to a skilled programmer or designer.

Avoid writing details in a design document that would be better placed as *comments* in the code.

Avoid writing details that can be *extracted automatically* from the code, such as the list of public methods.

9.7 Design of a Feature of the SimpleChat System

A. Purpose

This document describes important aspects of the implementation of the **`#block`**, **`#unblock`**, **`#whoiblock`** and **`#whoblocksme`** commands of the SimpleChat system.

B. General Priorities

Decisions in this document are made based on the following priorities (most important first): Maintainability, Usability, Portability, Efficiency

C. Outline of the design

Blocking information will be maintained in the **`ConnectionToClient`** objects. The various commands will update and query the data using **`setValue`** and **`getValue`**.

Design Example

D. Major design issue

Issue 1: Where should we store information regarding the establishment of blocking?

Option 1.1: Store the information in the **`ConnectionToClient`** object associated with the client requesting the block.

Option 1.2: Store the information in the **`ConnectionToClient`** object associated with the client that is being blocked.

Decision: Point 2.2 of the specification requires that we be able to block a client even if that client is not logged on. This means that we must choose option 1.1 since no **`ConnectionToClient`** will exist for clients that are logged off.

Design Example

E. Details of the design:

Client side:

The four new commands will be accepted by **`handleMessageFromClientUI`** and passed unchanged to the server.

Responses from the server will be displayed on the UI. There will be no need for **`handleMessageFromServer`** to understand that the responses are replies to the commands.

Design Example

Server side:

Method **`handleMessageFromClient`** will interpret **`#block`** commands by adding a record of the block in the data associated with the originating client.

This method will modify the data in response to **`#unblock`**.

The information will be stored by calling **`setValue("blockedUsers", arg)`**

where **`arg`** is a **`Vector`** containing the names of the blocked users.

Method **`handleMessageFromServerUI`** will also have to have an implementation of **`#block`** and **`#unblock`**.

These will have to save the blocked users as elements of a new instance variable declared thus: **`Vector blockedUsers`**;

Design Example

The implementations of **`#whoiblock`** in **`handleMessageFromClient`** and **`handleMessageFromServerUI`** will straightforwardly process the contents of the vectors.

For **`#whoblocksme`**, a new method will be created in the server class that will be called by both **`handleMessageFromClient`** and **`handleMessageFromServerUI`**.

This will take a single argument (the name of the initiating client, or else 'SERVER').

It will check all the **`blockedUsers`** vectors of the connected clients and also the **`blockedUsers`** instance variable for matching clients.

Design example

The **`#forward`**, **`#msg`** and **`#private`** commands will be modified as needed to reflect the specifications.

Each of these will each examine the relevant **`blockedUsers`** vectors and take appropriate action.

9.8 Difficulties and Risks in Design

Like modelling, design is a skill that requires considerable experience

- *—Individual software engineers should not attempt the design of large systems*
- *—Aspiring software architects should actively study designs of other systems*

Poor designs can lead to expensive maintenance

- *—Ensure you follow the principles discussed in this chapter*

Difficulties and Risks in Design

It requires constant effort to ensure a software system's design remains good throughout its life

- *—Make the original design as flexible as possible so as to anticipate changes and extensions.*
- *—Ensure that the design documentation is usable and at the correct level of detail*
- *—Ensure that change is carefully managed*

Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapter 10:

Testing and Inspecting to Ensure High Quality

10.1 Basic definitions

- A *failure* is an unacceptable behaviour exhibited by a system
  - —The frequency of failures measures the *reliability*
  - —An important design objective is to achieve a very low failure rate and hence high reliability.
  - —A failure can result from a violation of an ex*plicit* or *implicit* requirement
- A *defect* is a flaw in any aspect of the system that contributes, or may potentially contribute, to the occurrence of one or more failures
  - —It might take several defects to cause a particular failure
- An *error* is a slip-up or inappropriate decision by a software developer that leads to the introduction of a defect

10.2 Effective and Efficient Testing

To test *effectively*, you must use a strategy that uncovers as many defects as possible.

To test *efficiently*, you must find the largest possible number of defects using the fewest possible tests

- Testing is like detective work:
  - —The tester must try to understand how programmers and designers think, so as to better find defects.
  - —The tester must not leave anything uncovered, and must be suspicious of everything.
  - —It does not pay to take an excessive amount of time; tester has to be *efficient*.

Black-box testing

Testers provide the system with inputs and observe the outputs

- They can see none of:
  - —The source code
  - —The internal data
  - —Any of the design documentation describing the system's internals

Glass-box testing

Also called 'white-box' or 'structural' testing

Testers have access to the system design

- They can
  - —Examine the design documents
  - —View the code
  - —Observe at run time the steps taken by algorithms and their internal data
- Individual programmers often informally employ glass-box testing to verify their own code

Equivalence classes

- It is inappropriate to test by *brute force*, using *every possible* input value
  - —Takes a huge amount of time
  - —Is impractical
  - —Is pointless!
- You should divide the possible inputs into groups which you believe will be treated similarly by all algorithms.
  - —Such groups are called *equivalence classes*.
  - —A tester needs only to run one test per equivalence class
  - —The tester has to
    - understand the required input,
    - appreciate how the software may have been designed

Examples of equivalence classes

- Valid input is a month number (1-12)
  - —Equivalence classes are: [-∞..0], [1..12], [13.. ∞]
- Valid input is one of ten strings representing a type of fuel
  - —Equivalence classes are
    - 10 classes, one for each string
    - A class representing all other strings

Combinations of equivalence classes

- Combinatorial explosion means that you cannot realistically test every possible system-wide equivalence class.
  - —If there are 4 inputs with 5 possible values there are 5^4 (i.e.625) possible system-wide equivalence classes.
- You should first make sure that at least one test is run with every equivalence class of every individual input.
- You should also test all combinations where an input is likely to *affect the interpretation* of another.
- You should test a few other random combinations of equivalence classes.

Example equivalence class combinations

- One valid input is either 'Metric' or 'US/Imperial'
  - —Equivalence classes are:
    - Metric, US/Imperial, Other
- Another valid input is maxumum speed: 1 to 750 km/h or 1 to 500 mph
  - —Validity depends on whether metric or US/imperial
  - —Equivalence classes are:
    - [-∞..0], [1..500], [501..750], [751.. ∞]
- Some test combinations
  - Metric, [1..500] valid
  - US/Imperial, [501..750] invalid
  - Metric, [501..750] valid
  - Metric, [501..750] valid

Testing at boundaries of equivalence classes

- More errors in software occur at the boundaries of equivalence classes
- The idea of equivalence class testing should be expanded to specifically test values at the extremes of each equivalence class
  - —E.g. The number 0 often causes problems
- *E.g.*: If the valid input is a month number (1-12)
  - —Test equivalence classes as before
  - —Test 0, 1, 12 and 13 as well as very large positive and negative values

Detecting specific categories of defects

A tester must try to uncover any defects the other software engineers might have introduced.

- This means designing tests that explicitly try to catch a range of specific types of defects that commonly occur

10.3 Defects in Ordinary Algorithms

Incorrect logical conditions

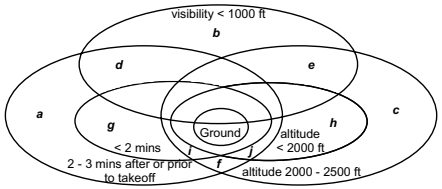
- *Defect*:
  - —The logical conditions that govern looping and if-then-else statements are wrongly formulated.
- *Testing strategy*:
  - —Use equivalence class and boundary testing.
  - —Consider as an input each variable used in a rule or logical condition.

Example of incorrect logical conditions defect

What is the hard-to-find defect in the following code?

- The landing gear must be deployed whenever the plane is within 2 minutes from landing or takeoff, or within 2000 feet from the ground. If visibility is less than 1000 feet, then the landing gear must be deployed whenever the plane is within 3 minutes from landing or lower than 2500 feet

```
if(!landingGearDeployed &&
  (min(now-takeoffTime,estLandTime-now))<
    (visibility < 1000 ? 180 :120) ||
  relativeAltitude <
    (visibility < 1000 ? 2500 :2000)
  )
{
 throw
   new LandingGearException();
}
```



Defects in Ordinary Algorithms

Performing a calculation in the wrong part of a control construct

- *Defect*:
  - —The program performs an action when it should not, or does not perform an action when it should.
  - —Typically caused by inappropriately excluding or including the action from a loop or a if construct.
- *Testing strategies*:
  - —Design tests that execute each loop zero times, exactly once, and more than once.
  - —Anything that could happen while looping is made to occur on the first, an intermediate, and the last iteration.

Example of performing a calculation in the wrong part of a control construct

```
while(j<maximum)
{
  k=someOperation(j);
  j++;
}
if(k==-1) signalAnError();
```


Defects in Ordinary Algorithms

Not terminating a loop or recursion

- *Defect*:
  - —A loop or a recursion does not always terminate, i.e. it is 'infinite'.
- *Testing strategies*:
  - —Analyse what causes a repetitive action to be stopped.
  - —Run test cases that you anticipate might not be handled correctly.

Defects in Ordinary Algorithms

Not setting up the correct preconditions for an algorithm

- *Defect*:
  - —*Preconditions* state what must be true before the algorithm should be executed.
  - —A defect would exist if a program proceeds to do its work, even when the preconditions are not satisfied.
- *Testing strategy*:
  - —Run test cases in which each precondition is not satisfied.

Defects in Ordinary Algorithms

Not handling null conditions

- *Defect*:
  - —A *null condition* is a situation where there normally are one or more data items to process, but sometimes there are none.
  - —It is a defect when a program behaves abnormally when a null condition is encountered.
- *Testing strategy*:
  - —Brainstorm to determine unusual conditions and run appropriate tests.

Defects in Ordinary Algorithms

Not handling singleton or non-singleton conditions

- *Defect*:
  - —A *singleton condition* occurs when there is normally *more than one* of something, but sometimes there is only one.
  - —A *non-singleton condition* is the inverse.
  - —Defects occur when the unusual case is not properly handled.
- *Testing strategy*:
  - —Brainstorm to determine unusual conditions and run appropriate tests.

Defects in Ordinary Algorithms

Off-by-one errors

- *Defect*:
  - —A program inappropriately adds or subtracts one.
  - —Or loops one too many times or one too few times.
  - —This is a particularly common type of defect.
- *Testing strategy*:
  - —Develop tests in which you verify that the program:
    - computes the correct numerical answer.
    - performs the correct number of iterations.

Example of off-by-one defect

```
for (i=1; i<arrayname.length; i++)
{
 /* do something */
}
```

Use `Iterators` to help eliminate these defects

```
while (iterator.hasNext())
{
 anOperation(++val);
}
```


Defects in Ordinary Algorithms

Operator precedence errors

- *Defect*:
  - —An operator precedence error occurs when a programmer omits needed parentheses, or puts parentheses in the wrong place.
  - —Operator precedence errors are often extremely obvious...
    - but can occasionally lie hidden until special conditions arise.
  - —E.g. If x*y+z should be x*(y+z) this would be hidden if z was normally zero.
- *Testing*:
  - —In software that computes formulae, run tests that anticipate such defects.

Defects in Ordinary Algorithms

Use of inappropriate standard algorithms

- *Defect*:
  - —An inappropriate standard algorithm is one that is unnecessarily inefficient or has some other property that is widely recognized as being bad.
- *Testing strategies*:
  - —The tester has to know the properties of algorithms and design tests that will determine whether any undesirable algorithms have been implemented.

Example of inappropriate standard algorithms

- An inefficient sort algorithm
  - —The most classical ‘bad’ choice of algorithm is sorting using a so-called ‘bubble sort’
- An inefficient search algorithm
  - —Ensure that the search time does not increase unacceptably as the list gets longer
  - —Check that the position of the searched item does not have a noticeable impact on search time.
- A non-stable sort
- A search or sort that is case sensitive when it should not be, or vice versa

10.4 Defects in Numerical Algorithms

Not using enough bits or digits

- *Defect*:
  - —A system does not use variables capable of representing the largest values that could be stored.
  - —When the capacity is exceeded, an unexpected exception is thrown, or the data stored is incorrect.
- *Testing strategies*:
  - —Test using very large numbers to ensure the system has a wide enough margin of error.

Defects in Numerical Algorithms

Not using enough places after the decimal point or significant figures

- *Defects*:
  - —A floating point value might not have the capacity to store enough significant figures.
  - —A fixed point value might not store enough places after the decimal point.
  - — A typical manifestation is excessive rounding.
- *Testing strategies*:
  - —Perform calculations that involve many significant figures, and large differences in magnitude.
  - —Verify that the calculated results are correct.

Defects in Numerical Algorithms

Ordering operations poorly so errors build up

- *Defect*:
  - —A large number does not store enough significant figures to be able to accurately represent the result.
- *Testing strategies*:
  - —Make sure the program works with inputs that have large positive and negative exponents.
  - —Have the program work with numbers that vary a lot in magnitude.
    - Make sure computations are still accurately performed.

Defects in Numerical Algorithms

Assuming a floating point value will be exactly equal to some other value

- *Defect*:
  - —If you perform an arithmetic calculation on a floating point value, then the result will very rarely be computed exactly.
  - —To test equality, you should always test if it is within a small range around that value.
- *Testing strategies*:
  - —Standard boundary testing should detect this type of defect.

Example of defect in testing floating value equality

Bad:

```
for (double d = 0.0; d != 10.0; d+=2.0) {...}
```

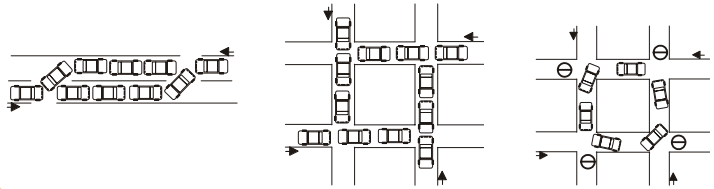
Better:

```
for (double d = 0.0; d < 10.0; d+=2.0) {...}
```


10.5 Defects in Timing and Co-ordination

Deadlock and livelock

- *Defects*:
  - —A deadlock is a situation where two or more threads are stopped, waiting for each other to do something.
    - The system is hung
  - —Livelock is similar, but now the system can do some computations, but can never get out of some states.

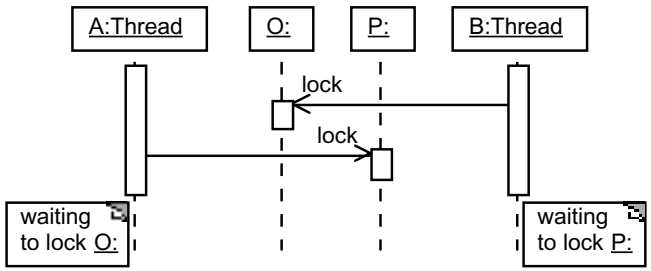


Defects in Timing and Co-ordination

Deadlock and livelock

- *Testing strategies*:
  - —Deadlocks and livelocks occur due to unusual combinations of conditions that are hard to anticipate or reproduce.
  - —It is often most effective to use *inspection* to detect such defects, rather than testing alone.
  - —However, when testing:
    - Vary the time consumption of different threads.
    - Run a large number of threads concurrently.
    - Deliberately deny resources to one or more threads.

Example of deadlock

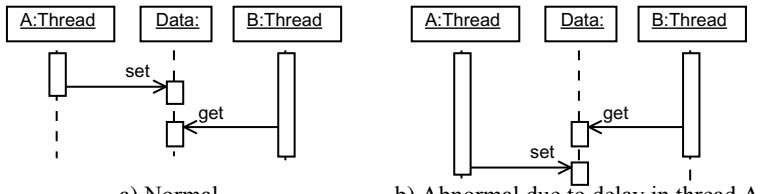


Defects in Timing and Co-ordination

Critical races

- *Defects*:
  - —One thread experiences a failure because another thread interferes with the 'normal' sequence of events.
- *Testing strategies*:
  - —It is particularly hard to test for critical races using black box testing alone.
  - —One possible, although invasive, strategy is to deliberately slow down one of the threads.
  - —Use inspection.

Example of critical race



a) Normal

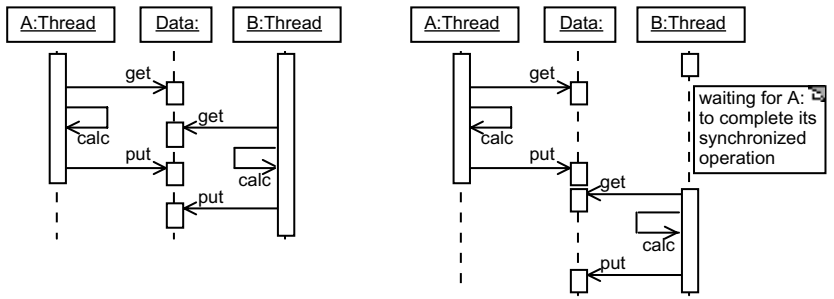
b) Abnormal due to delay in thread A

Semaphore and synchronization

Critical races can be prevented by *locking* data so that they cannot be accessed by other threads when they are not ready

- One widely used locking mechanism is called a *semaphore*.
- In Java, the **synchronized** keyword can be used.
  - —It ensures that no other thread can access an object until the synchronized method terminates.

Example of a synchronized method



a) Abnormal: The value put by thread A is immediately overwritten by the value put by thread B.

b) The problem has been solved by accessing the data using synchronized methods

10.6 Defects in Handling Stress and Unusual Situations

Insufficient throughput or response time on minimal configurations

- *Defect*:
  - —On a minimal configuration, the system's throughput or response time fail to meet requirements.
- *Testing strategy*:
  - —Perform testing using minimally configured platforms.

Defects in Handling Stress and Unusual Situations

Incompatibility with specific configurations of hardware or software

- *Defect*:
  - —The system fails if it is run using particular configurations of hardware, operating systems and external libraries.
- *Testing strategy*:
  - —Extensively execute the system with all possible configurations that might be encountered by users.

Defects in Handling Stress and Unusual Situations

Defects in handling peak loads or missing resources

- *Defects*:
  - —The system does not gracefully handle resource shortage.
  - —Resources that might be in short supply include:
    - memory, disk space or network bandwidth, permission.
  - —The program being tested should report the problem in a way the user will understand.
- *Testing strategies*:
  - —Devise a method of denying the resources.
  - —Run a very large number of copies of the program being tested, all at the same time.

Defects in Handling Stress and Unusual Situations

Inappropriate management of resources

- *Defect*:
  - —A program uses certain resources but does not make them available when it no longer needs them.
- *Testing strategy*:
  - —Run the program intensively in such a way that it uses many resources, relinquishes them and then uses them again repeatedly.

Defects in Handling Stress and Unusual Situations

Defects in the process of recovering from a crash

- *Defects*:
  - —Any system will undergo a sudden failure if its hardware fails, or if its power is turned off.
  - —It is a defect if the system is left in an unstable state and hence is unable to fully recover.
  - —It is also a defect if a system does not correctly deal with the crashes of related systems.
- *Testing strategies*:
  - —Kill a program at various times during execution.
  - —Try turning the power off, however operating systems themselves are often intolerant of doing that.

10.7 Documentation defects

- *Defect*:
  - —The software has a defect if the user manual, reference manual or on-line help:
    - gives incorrect information
    - fails to give information relevant to a problem.
- *Testing strategy*:
  - —Examine all the end-user documentation, making sure it is correct.
  - —Work through the use cases, making sure that each of them is adequately explained to the user.

10.8 Writing Formal Test Cases and Test Plans

A *test case* is an explicit set of instructions designed to detect a particular class of defect in a software system.

- A test case can give rise to many tests.
- Each test is a particular running of the test case on a particular version of the system.

Test plans

A *test plan* is a document that contains a complete set of test cases for a system

- —Along with other information about the testing process.
- The test plan is one of the standard forms of documentation.
- If a project does not have a test plan:
  - —Testing will inevitably be done in an ad-hoc manner.
  - —Leading to poor quality software.
- The test plan should be written long before the testing starts.
- You can start to develop the test plan once you have developed the requirements.

Information to include in a formal test case

A. Identification and classification:
- —Each test case should have a number, and may also be given a descriptive title.
- —The system, subsystem or module being tested should also be clearly indicated.
- —The importance of the test case should be indicated.

B. Instructions:
- —Tell the tester exactly what to do.
- —The tester should not normally have to refer to any documentation in order to execute the instructions.

C. Expected result:
- —Tells the tester what the system should do in response to the instructions.
- —The tester reports a failure if the expected result is not encountered.

D. Cleanup (when needed):
- —Tells the tester how to make the system go 'back to normal' or shut down after the test.

Levels of importance of test cases

- Level 1:
  - —First pass critical test cases.
  - —Designed to verify the system runs and is safe.
  - —No further testing is possible.
- Level 2:
  - —General test cases.
  - —Verify that day-to-day functions correctly.
  - —Still permit testing of other aspects of the system.
- Level 3:
  - —Detailed test cases.
  - —Test requirements that are of lesser importance.
  - —The system functions most of the time but has not yet met quality objectives.

Determining test cases by enumerating attributes

It is important that the test cases test every aspect of the requirements.

- Each detail in the requirements is called an *attribute*.
  - —An attribute can be thought of as something that is testable.
  - —A good first step when creating a set of test cases is to *enumerate* the attributes.
  - —A way to enumerate attributes is to circle all the important points in the requirements document.
- However there are often many attributes that are *implicit*.

10.9 Strategies for Testing Large Systems

Big bang testing versus integration testing

- In *big bang* testing, you take the entire system and test it as a unit
- A better strategy in most cases is *incremental testing*:
  - —You test each individual subsystem in isolation
  - —Continue testing as you add more and more subsystems to the final product
  - —Incremental testing can be performed *horizontally* or *vertically*, depending on the architecture
    - Horizontal testing can be used when the system is divided into separate sub-applications

Top down testing

- Start by testing just the user interface.
- The underlying functionality are simulated by *stubs*.
  - —Pieces of code that have the same interface as the lower level functionality.
  - —Do not perform any real computations or manipulate any real data.
- Then you work downwards, integrating lower and lower layers.
- The big drawback to top down testing is the cost of writing the stubs.

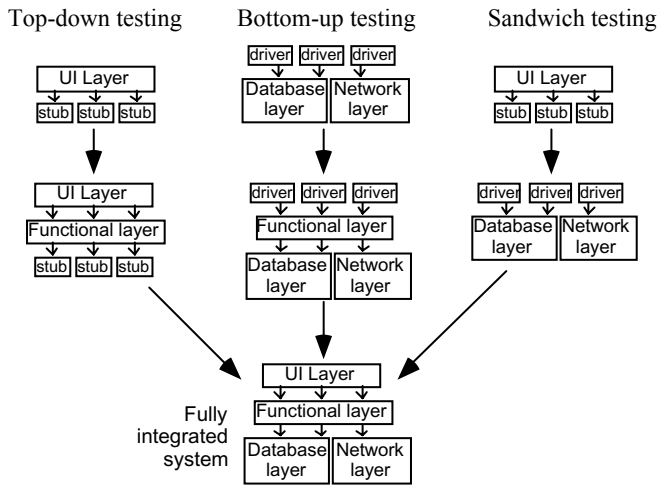
Bottom-up testing

- Start by testing the very lowest levels of the software.
- You needs *drivers* to test the lower layers of software.
  - —Drivers are simple programs designed specifically for testing that make calls to the lower layers.
- Drivers in bottom-up testing have a similar role to stubs in top-down testing, and are time-consuming to write.

Sandwich testing

- Sandwich testing is a hybrid between bottom-up and top down testing.
- Test the user interface in isolation, using stubs.
- Test the very lowest level functions, using drivers.
- When the complete system is integrated, only the middle layer remains on which to perform the final set of tests.

Vertical strategies for incremental integration testing



The test-fix-test cycle

When a failure occurs during testing:

- Each failure report is entered into a failure tracking system.
- It is then screened and assigned a priority.
- Low-priority failures might be put on a *known bugs list* that is included with the software's *release notes*.
- Some failure reports might be merged if they appear to result from the same defects.
- Somebody is assigned to investigate a failure.
- That person tracks down the defect and fixes it.
- Finally a new version of the system is created, ready to be tested again.

The ripple effect

There is a high probability that the efforts to remove the defects may have actually added new defects

- The maintainer tries to fix problems without fully understanding the ramifications of the changes
- The maintainer makes ordinary human errors
- The system *regresses* into a more and more failure-prone state

Regression testing

- It tends to be far too expensive to re-run every single test case every time a change is made to software.
- Hence only a subset of the previously-successful test cases is actually re-run.
- This process is called *regression testing.*
  —The tests that are re-run are called regression tests.
- Regression test cases are carefully selected to cover as much of the system as possible.

The "law of conservation of bugs":

- *The number of bugs remaining in a large system is proportional to the number of bugs already fixed*

Deciding when to stop testing

- All of the level 1 test cases must have been successfully executed.
- Certain pre-defined percentages of level 2 and level 3 test cases must have been executed successfully.
- The targets must have been achieved and are maintained for at least two cycles of 'builds'.
  —A *build* involves compiling and integrating all the components.
  —Failure rates can fluctuate from build to build as:
  - Different sets of regression tests are run.
  - New defects are introduced.

The roles of people involved in testing

- The first pass of unit and integration testing is called *developer testing*.
  - —Preliminary testing performed by the software developers who do the design.
- *Independent testing* is performed by a separate group.
  - —They do not have a vested interest in seeing as many test cases pass as possible.
  - —They develop specific expertise in how to do good testing, and how to use testing tools.

Testing performed by users and clients

- *Alpha testing*
  - —Performed by the user or client, but under the supervision of the software development team.
- *Beta testing*
  - —Performed by the user or client in a normal work environment.
  - —Recruited from the potential user population.
  - —An *open beta release* is the release of low-quality software to the general population.
- *Acceptance testing*
  - —Performed by users and customers.
  - —However, the customers do it on their own initiative.

10.10 Inspections

An inspection is an activity in which one or more people systematically

- Examine source code or documentation, looking for defects.
- Normally, inspection involves a meeting...
  - —Although participants can also inspect alone at their desks.

Roles on inspection teams

- The *author*
- The *moderator*.
  - —Calls and runs the meeting.
  - —Makes sure that the general principles of inspection are adhered to.
- The *secretary*.
  - —Responsible for recording the defects when they are found.
  - —Must have a thorough knowledge of software engineering.
- *Paraphrasers*.
  - —Step through the document explaining it in their own words.

Principles of inspecting

- Inspect the most important documents of all types
  - —code, design documents, test plans and requirements
- Choose an effective and efficient inspection team
  - —between two and five people
  - —Including experienced software engineers
- Require that participants prepare for inspections
  - —They should study the documents prior to the meeting and come prepared with a list of defects
- Only inspect documents that are ready
  - —Attempting to inspect a very poor document will result in defects being missed

Principles of inspecting

- Avoid discussing how to fix defects
  - —Fixing defects can be left to the author
- Avoid discussing style issues
  - —Issues like are important, but should be discussed separately
- Do not rush the inspection process
  - —A good speed to inspect is
    - 200 lines of code per hour (including comments)
    - or ten pages of text per hour

Principles of inspecting

- Avoid making participants tired
  - —It is best not to inspect for more than two hours at a time, or for more than four hours a day
- Keep and use logs of inspections
  - —You can also use the logs to track the quality of the design process
- Re-inspect when changes are made
  - —You should re-inspect any document or code that is changed more than 20%

A peer-review process

Managers are normally not involved

- This allows the participants to express their criticisms more openly, not fearing repercussions
- The members of an inspection team should feel they are all working together to create a better document
- Nobody should be blamed

Conducting an inspection meeting

1. The moderator calls the meeting and distributes the documents.
2. The participants prepare for the meeting in advance.
3. At the start of the meeting, the moderator explains the procedures and verifies that everybody has prepared.
4. Paraphrasers take turns explaining the contents of the document or code, without reading it verbatim.
   - Requiring that the paraphraser not be the author ensures that the paraphraser say what he or she *sees*, not what the author *intended* to say.
5. Everybody speaks up when they notice a defect.

Inspecting compared to testing

- Both testing and inspection rely on different aspects of human intelligence.
- Testing can find defects whose consequences are obvious but which are buried in complex code.
- Inspecting can find defects that relate to maintainability or efficiency.
- The chances of mistakes are reduced if both activities are performed.

Testing or inspecting, which comes first?

- It is important to inspect software *before* extensively testing it.
- The reason for this is that inspecting allows you to quickly get rid of many defects.
- If you test first, and inspectors recommend that redesign is needed, the testing work has been wasted.
  - —There is a growing consensus that it is most efficient to inspect software before *a*ny testing is done.
- Even before developer testing

10.11 Quality Assurance in General

Root cause analysis

- Determine whether problems are caused by such factors as
  - —Lack of training
  - —Schedules that are too tight
  - —Building on poor designs or reusable technology

Measure quality and strive for continual improvement

Things you can measure regarding the quality of a software product, and indirectly of the quality of the process

- The number of failures encountered by users.
- The number of failures found when testing a product.
- The number of defects found when inspecting a product.
- The percentage of code that is reused.
  —More is better, but don't count clones.
- The number of questions posed by users to the help desk.
  —As a measure of usability and the quality of documentation.

Post-mortem analysis

Looking back at a project after it is complete, or after a release,

- You look at the design and the development process
- Identify those aspects which, with benefit of hindsight, you could have done better
- You make plans to do better next time

Process standards

The personal software process (PSP):

- Defines a disciplined approach that a developer can use to improve the quality and efficiency of his or her personal work.
- One of the key tenets is personally inspecting your own work.

The team software process (TSP):

- Describes how teams of software engineers can work together effectively.

The software capability maturity model (CMM):

- Contains five levels, Organizations start in level 1, and as their processes become better they can move up towards level 5.

ISO 9000-2:

- An international standard that lists a large number of things an organization should do to improve their overall software process.

10.12 Detailed Example: Test cases for Phase 2 of the SimpleChat

General Setup for Test Cases in the 2000 Series
System: SimpleChat/OCSF Phase: 2

Instructions:

1. Install Java, minimum release 1.2.0, on Windows 95, 98 or ME.
2. Install Java, minimum release 1.2.0, on Windows NT or 2000.
3. Install Java, minimum release 1.2.0, on a Solaris system.
4. Install the SimpleChat - Phase 2 on each of the above platforms.

Test cases for Phase 2 of the SimpleChat

Test Case 2001
System: SimpleChat Phase: 2
Server startup check with default arguments
Severity: 1

Instructions:
1. At the console, enter: **`java EchoServer`**.

Expected result:
1. The server reports that it is listening for clients by displaying the following message:
   `Server listening for clients on port 5555`
2. The server console waits for user input.

Cleanup:
1. Hit CTRL+C to kill the server.

Test cases for Phase 2 of the SimpleChat

Test Case 2002
System: SimpleChat Phase: 2
Client startup check without a login
Severity: 1

Instructions:
1. At the console, enter: **`java ClientConsole`**.

Expected result:
1. The client reports it cannot connect without a login by displaying:
   `ERROR - No login ID specified. Connection aborted.`
2. The client terminates.

Cleanup: (if client is still active)
1. Hit CTRL+C to kill the client.

Test cases for Phase 2 of the SimpleChat

Test Case 2003
System: SimpleChat Phase: 2
Client startup check with a login and without a server
Severity: 1

Instructions:
1. At the console, enter: **`java ClientConsole <loginID>`**
   where **`<loginID>`** is the name you wish to be identified by.

Expected result:
1. The client reports it cannot connect to a server by displaying:
   `Cannot open connection. Awaiting command.`
2. The client waits for user input

Cleanup: (if client is still active)
1. Hit CTRL+C to kill the client.

Test cases for Phase 2 of the SimpleChat

Test Case 2007
System: SimpleChat Phase: 2
Server termination command check
Severity: 2

Instructions:
1. Start a server (Test Case 2001 instruction 1) using default arguments.
2. Type **`#quit`** into the server's console.

Expected result:
1. The server quits.

Cleanup (If the server is still active):
1. Hit CTRL+C to kill the server.

Test cases for Phase 2 of the SimpleChat

Test Case 2013
System: SimpleChat Phase: 2
Client host and port setup commands check
Severity: 2
Instructions:
1. Start a client without a server (Test Case 2003).
2. At the client's console, type **#sethost <newhost>** where
- **<newhost>** is the name of a computer on the network
3. At the client's console, type **#setport 1234**.

Expected result:
1. The client displays
Host set to: <newhost>
Port set to: 1234.
Cleanup:
1. Type **#quit** to kill the client.

Test cases for Phase 2 of the SimpleChat

Test Case 2019
System: SimpleChat Phase: 2
Different platform tests
Severity: 3

Instructions:
1. Repeat test cases 2001 to 2018 on Windows 95, 98, NT or 2000, and Solaris

Expected results:
1. The same as before.

10.13 Difficulties and Risks in Quality Assurance

It is very easy to forget to test some aspects of a software system:

—*'running the code a few times' is not enough.*

—*Forgetting certain types of tests diminishes the system's quality.*

There is a conflict between achieving adequate quality levels, and 'getting the product out of the door'

—*Create a separate department to oversee QA.*

—*Publish statistics about quality.*

—*Build adequate time for all activities.*

Difficulties and Risks in Quality Assurance

People have different abilities and knowledge when it comes to quality

—*Give people tasks that fit their natural personalities.*

—*Train people in testing and inspecting techniques.*

— *Give people feedback about their performance in terms of producing quality software.*

—*Have developers and maintainers work for several months on a testing team.*

Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapter 11:
Managing the Software Process

11.1 What is Project Management?

Project management encompasses all the activities needed to plan and execute a project:

- Deciding what needs to be done
- Estimating costs
- Ensuring there are suitable people to undertake the project
- Defining responsibilities
- Scheduling
- Making arrangements for the work

continued ...

What is Project Management?

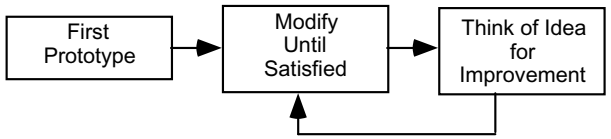
- Directing
- Being a technical leader
- Reviewing and approving decisions made by others
- Building morale and supporting staff
- Monitoring and controlling
- Co-ordinating the work with managers of other projects
- Reporting
- Continually striving to improve the process

11.2 Software Process Models

Software process models are general approaches for organizing a project into activities.

- Help the project manager and his or her team to decide:
  - —What work should be done;
  - —In what sequence to perform the work.
- The models should be seen as *aids to thinking*, not rigid prescriptions of the way to do things.
- Each project ends up with its own unique plan.

The opportunistic approach

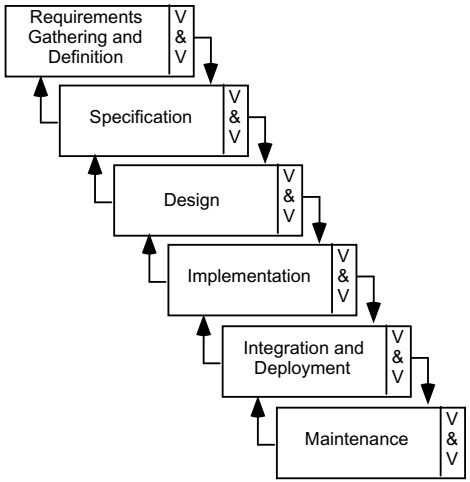


The opportunistic approach

… is what occurs when an organization does not follow good engineering practices.

- It does not acknowledge the importance of working out the requirements and the design before implementing a system.
- The design of software deteriorates faster if it is not well designed.
- Since there are no plans, there is nothing to aim towards.
- There is no explicit recognition of the need for systematic testing and other forms of quality assurance.
- The above problems make the cost of developing and maintaining software very high.

The waterfall model



The waterfall model

The classic way of looking at S.E. that accounts for the importance of requirements, design and quality assurance.

- The model suggests that software engineers should work in a series of stages.
- Before completing each stage, they should perform quality assurance (verification and validation).
- The waterfall model also recognizes, to a limited extent, that you sometimes have to step back to earlier stages.

Limitations of the waterfall model

The model implies that you should attempt to complete a given stage before moving on to the next stage

—Does not account for the fact that requirements constantly change.

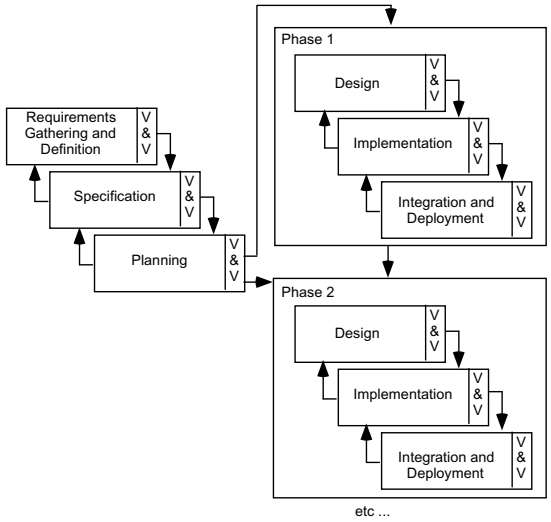
—It also means that customers can not use anything until the entire system is complete.

The model makes no allowances for prototyping.

It implies that you can get the requirements right by simply writing them down and reviewing them.

The model implies that once the product is finished, everything else is maintenance.

The phased-release model



The phased-release model

It introduces the notion of *incremental* development.

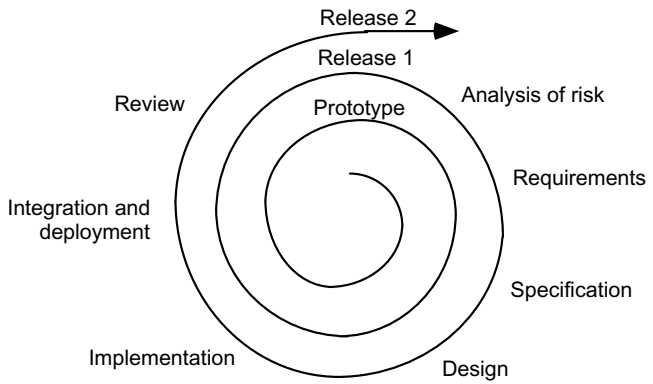
After requirements gathering and planning, the project should be broken into separate subprojects, or *phases*.

Each phase can be released to customers when ready.

Parts of the system will be available earlier than when using a strict waterfall approach.

However, it continues to suggest that all requirements be finalized at the start of development.

The spiral model

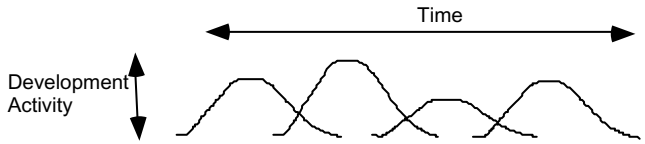


The spiral model

It explicitly embraces prototyping and an *iterative* approach to software development.

- Start by developing a small prototype.
- Followed by a mini-waterfall process, primarily to gather requirements.
- Then, the first prototype is reviewed.
- In subsequent loops, the project team performs further requirements, design, implementation and review.
- The first thing to do before embarking on each new loop is risk analysis.
- Maintenance is simply a type of on-going development.

The evolutionary model

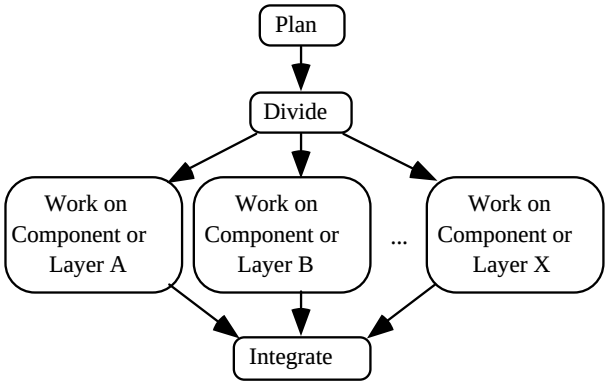


The evolutionary model

It shows software development as a series of hills, each representing a separate loop of the spiral.

- Shows that loops, or releases, tend to overlap each other.
- Makes it clear that development work tends to reach a peak, at around the time of the deadline for completion.
- Shows that each prototype or release can take
  - —different amounts of time to deliver;
  - —differing amounts of effort.

The concurrent engineering model



The concurrent engineering model

It explicitly accounts for the divide and conquer principle.

Each team works on its own component, typically following a spiral or evolutionary approach.

There has to be some initial planning, and periodic integration.

Choosing a process model

From the waterfall model:

—Incorporate the notion of stages.

From the phased-release model:

—Incorporate the notion of doing some initial high-level analysis, and then dividing the project into releases.

From the spiral model:

—Incorporate prototyping and risk analysis.

From the evolutionary model:

—Incorporate the notion of varying amounts of time and work, with overlapping releases.

From the concurrent engineering:

—Incorporate the notion of breaking the system down into components and developing them in parallel.

Reengineering

Periodically project managers should set aside some time to re-engineer part or all of the system

The extent of this work can vary considerably:

—Cleaning up the code to make it more readable.

—Completely replacing a layer.

—*Re-factoring* part of the design.

In general, the objective of a re-engineering activity is to increase maintainability.

11.3 Cost estimation

To estimate how much software-engineering time will be required to do some work.

Elapsed time

—The difference in time from the start date to the end date of a task or project.

Development effort

—The amount of labour used in *person-months* or *person-days.*

—To convert an estimate of development effort to an amount of money:

You multiply it by the *weighted average cost* (*burdened* cost) of employing a software engineer for a month (or a day).

Principles of effective cost estimation

Principle 1: Divide and conquer.

To make a better estimate, you should divide the project up into individual subsystems.

Then divide each subsystem further into the activities that will be required to develop it.

Next, you make a series of detailed estimates for each individual activity.

And sum the results to arrive at the grand total estimate for the project.

Principles of effective cost estimation

Principle 2: Include all activities when making estimates.

The time required for *all* development activities must be taken into account.

Including:

- Prototyping
- Design
- Inspecting
- Testing
- Debugging
- Writing user documentation
- Deployment.

Principles of effective cost estimation

Principle 3: Base your estimates on past experience combined with knowledge of the current project.

If you are developing a project that has many similarities with a past project:

— You can expect it to take a similar amount of work.

Base your estimates on the *personal judgement* of your experts

or

Use *algorithmic models* developed in the software industry as a whole by analyzing a wide range of projects.

—They take into account various aspects of a project's size and complexity, and provide formulas to compute anticipated cost.

Algorithmic models

Allow you to systematically estimate development effort.

Based on an estimate of some other factor that you can measure, or that is easier to estimate:

—The number of use cases

—The number of distinct requirements

—The number of classes in the domain model

—The number of widgets in the prototype user interface

—An estimate of the number of lines of code

Algorithmic models

A typical algorithmic model uses a formula like the following:

—COCOMO:

$$E = a + bN^c$$

—Functions Points:

$$S = W_1F_1 + W_2F_2 + W_3F_3 + \ldots$$

Principles of effective cost estimation

Principle 4: Be sure to account for *differences* when extrapolating from other projects.

- Different software developers
- Different development processes and maturity levels
- Different types of customers and users
- Different schedule demands
- Different technology
- Different technical complexity of the requirements
- Different domains
- Different levels of requirement stability

Principles of effective cost estimation

Principle 5: Anticipate the worst case and plan for contingencies.

- Develop the most critical use cases first
  - —If the project runs into difficulty, then the critical features are more likely to have been completed
- Make three estimates:
  - —Optimistic (O)
    - Imagining a everything going perfectly
  - —Likely (L)
    - Allowing for typical things going wrong
  - —Pessimistic
    - Accounting for everything that could go wring

Principles of effective cost estimation

Principle 6: Combine multiple independent estimates.

- Use several different techniques and compare the results.
- If there are discrepancies, analyze your calculations to discover what factors causing the differences.
- Use the Delphi technique.
  - —Several individuals initially make cost estimates in private.
  - —They then share their estimates to discover the discrepancies.
  - —Each individual repeatedly adjusts his or her estimates until a consensus is reached.

Principles of effective cost estimation

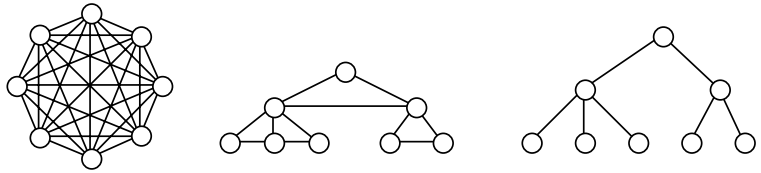
Principle 7: Revise and refine estimates as work progresses

- As you add detail.
- As the requirements change.
- As the risk management process uncovers problems.

11.4 Building Software Engineering Teams

Software engineering is a human process.

- Choosing appropriate people for a team, and assigning roles and responsibilities to the team members, is therefore an important project management skill
- Software engineering teams can be organized in many different ways



Software engineering teams

Egoless team:

- In such a team everybody is equal, and the team works together to achieve a common goal.
- Decisions are made by consensus.
- Most suited to difficult projects with many technical challenges.

Software engineering teams

Hierarchical manager-subordinate structure:

- Each individual reports to a manager and is responsible for performing the tasks delegated by that manager.
- Suitable for large projects with a strict schedule where everybody is well-trained and has a well-defined role.
- However, since everybody is only responsible for their own work, problems may go unnoticed.

Software engineering teams

Chief programmer team:

- Midway between egoless and hierarchical.
- The chief programmer leads and guides the project.
- He or she consults with, and relies on, individual specialists.

Choosing an effective size for a team

For a given estimated development effort, in person months, there is an optimal team size.

- —Doubling the size of a team will not halve the development time.

Subsystems and teams should be sized such that the total amount of required knowledge and exchange of information is reduced.

For a given project or project iteration, the number of people on a team will not be constant.

You can not generally add people if you get behind schedule, in the hope of catching up.

Skills needed on a team

- Architect
- Project manager
- Configuration management and build specialist
- User interface specialist
- Technology specialist
- Hardware and third-party software specialist
- User documentation specialist
- Tester

11.5 Project Scheduling and Tracking

Scheduling is the process of deciding:

- —In what sequence a set of activities will be performed.
- —When they should start and be completed.

Tracking is the process of determining how well you are sticking to the cost estimate and schedule.

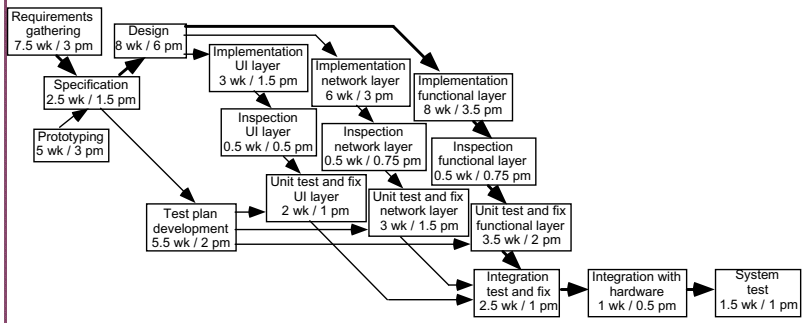
PERT charts

A PERT chart shows the sequence in which tasks must be completed.

In each node of a PERT chart, you typically show the elapsed time and effort estimates.

The *critical path* indicates the minimum time in which it is possible to complete the project.

Example of a PERT chart



Gantt charts

A Gantt chart is used to graphically present the start and end dates of each software engineering task

One axis shows time.

The other axis shows the activities that will be performed.

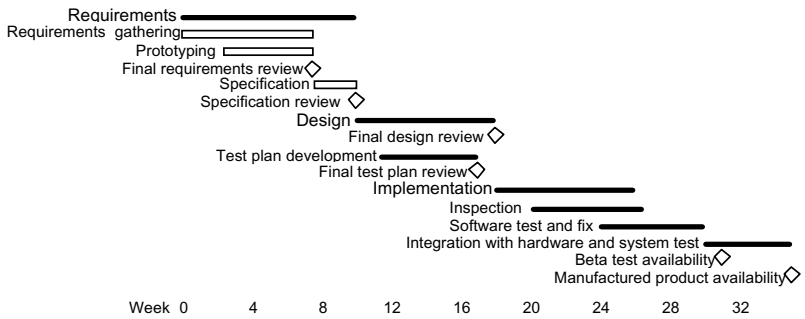
The black bars are the top-level tasks.

The white bars are subtasks

The diamonds are *milestones:*

—Important deadline dates, at which specific events may occur

Example of a Gantt chart



Earned value

Earned value is the amount of work completed, measured according to the *budgeted* effort that the work was supposed to consume.

It is also called the *budgeted cost of work performed*.

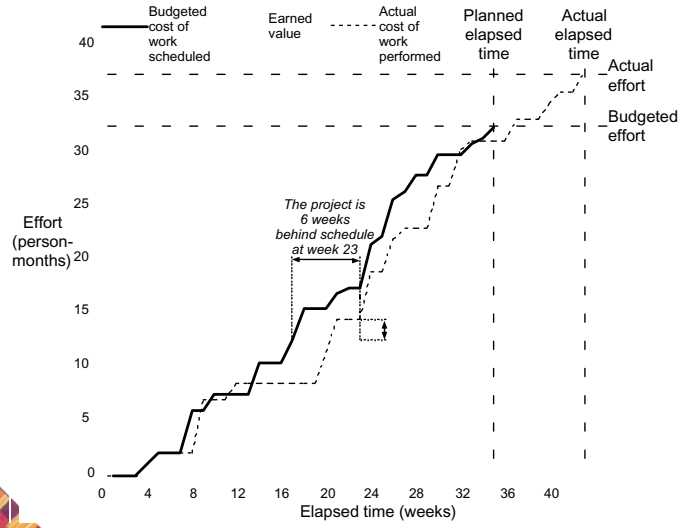
As each task is completed, the number of person-months originally planned for that task is added to the earned value of the project.

Earned value charts

An earned value chart has three curves:

- The budgeted cost of the work scheduled.
- The earned value.
- The actual cost of the work performed so far.

Example of an earned value chart



11.6 Contents of a Project Plan

A. Purpose
B. Background information
C. Processes to be used
D. Subsystems and planned releases
E. Risks and challenges
F. Tasks
G. Cost estimates
H. Team
I. Schedule and milestones

11.7 Difficulties and Risks in Project Management

Accurately estimating costs is a constant challenge

—*Follow the cost estimation guidelines.*

It is very difficult to measure progress and meet deadlines

—*Improve your cost estimation skills so as to account for the kinds of problems that may occur.*

—*Develop a closer relationship with other members of the team.*

—*Be realistic in initial requirements gathering, and follow an iterative approach.*

—*Use earned value charts to monitor progress.*

Difficulties and Risks in Project Management

It is difficult to deal with lack of human resources or technology needed to successfully run a project

—*When determining the requirements and the project plan, take into consideration the resources available.*

—*If you cannot find skilled people or suitable technology then you must limit the scope of your project.*

Difficulties and Risks in Project Management

Communicating effectively in a large project is hard

—*Take courses in communication, both written and oral.*

—*Learn how to run effective meetings.*

—*Review what information everybody should have, and make sure they have it.*

—*Make sure that project information is readily available.*

—*Use 'groupware' technology to help people exchange the information they need to know*

Difficulties and Risks in Project Management

It is hard to obtain agreement and commitment from others

—*Take courses in negotiating skills and leadership.*

—*Ensure that everybody understands*

- *The position of everybody else.*
- *The costs and benefits of each alternative.*
- *The rationale behind any compromises.*

—*Ensure that everybody's proposed responsibility is clearly expressed.*

—*Listen to everybody's opinion, but take assertive action, when needed, to ensure progress occurs.*